VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

UMĚLÁ INTELIGENCE NA PLATROFMĚ NVIDIA JETSON
ARTIFICIAL INTELLIGENCE ON NVIDIA JETSON PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Lukáš Batelka**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Petr Blaha, Ph.D.**
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Lukáš Batelka ***ID:*** 211414
Ročník: 3 ***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Umělá inteligence na platrofmě nVIDIA Jetson

POKYNY PRO VYPRACOVÁNÍ:

1. Seznamte se se současným stavem poznání v oblasti implementace umělé inteligence ve vestavných zařízeních.
2. Seznamte se s nástroji pro trénování umělých neuronových sítí (Deep Learning Toolbox, TensorFlow, Keras).
3. Natrénujte umělou neuronovou síť na předložená označkovaná data ze simulačních experimentů.
4. Proveďte implementaci natrénované sítě na platformě nVIDIA Jetson. Při implementaci se zaměřte na optimalizaci sítě.
5. Porovnejte výsledky klasifikace na PC a na platformě nVIDIA Jetson.

DOPORUČENÁ LITERATURA:

[1] Ertel, W.: Introduction to Artificial Intelligence. Springer International Publishing, 2017. ISBN 978-3-319-58-87-4.

další dle doporučení vedoucího práce

Termín zadání: 8.2.2021 ***Termín odevzdání:*** 24.5.2021

Vedoucí práce: doc. Ing. Petr Blaha, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

Abstrakt

Cílem této bakalářské práce je vytvoření, naučení a implementace umělé neuronové sítě ve vestavném zařízení NVIDIA Jetson Nano. V první části práce je popsán současný stav implementace umělé inteligence ve vestavných zařízeních. Následující část popisuje nástroje pro vývoj umělých neuronových sítí a možnosti jejich implementace v zařízení. Tyto nástroje jsou dále v práci použity pro vytvoření a natrénování umělé neuronové sítě, která má za cíl detekovat poruchu v předzpracovaných datech z měření na synchronním elektromotoru. Nakonec je popsán způsob provedené optimalizace natrénované neuronové sítě. Dosažené výsledky jsou shrnuty v závěru práce.

Klíčová slova

NVIDIA Jetson Nano, detekce poruchy, umělá neuronová síť, CUDA, kvantizace, embedded AI, Keras, Python

Abstract

The aim of this bachelor thesis is to design, train and implement an artificial neural network in an NVIDIA Jetson Nano embedded device. The first part of the thesis describes the current state of the art of implementing artificial intelligence in embedded devices. The following section describes the tools for developing artificial neural networks and the possibilities of implementing them in a Jetson Nano device. These tools are further used in the thesis to create and train an artificial neural network to detect a fault in preprocessed measurement data on a synchronous electric motor. Finally, the optimization of the trained neural network is described. The achieved results are summarized in the conclusion of the paper.

Keywords

NVIDIA Jetson Nano, fault detection, artificial neural network, CUDA, quantization, embedded AI, Keras, Python

Bibliografická citace

BATELKA, Lukáš. Umělá inteligence na platrofmě nVIDIA Jetson. Brno, 2021. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/134794. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Petr Blaha.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	*Lukáš Batelka*
VUT ID studenta:	*211414*
Typ práce:	*Bakalářská práce*
Akademický rok:	*2020/21*
Téma závěrečné práce:	*Umělá inteligence na platrofmě nVIDIA Jetson*

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne: 24. května 2021 ----------

podpis autora

Poděkování

Děkuji vedoucímu bakalářské práce doc. Ing. Petru Blahovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne: 24. května 2021 ----------

podpis autora

Obsah

SEZNAM SYMBOLŮ A ZKRATEK

Zkratky:

API	Application Programming Interface (rozhraní pro programování aplikací)
BSD	Berkeley Software Distribution (licence pro svobodný software vytvořená na University of California,)
CPU	Central Processing Unit (centrální procesorová jednotka)
CSV	Comma-Separated Values (hodnoty oddělené čárkami)
CUDA	Compute Unified Device Architecture (architektura jednotného výpočetního zařízení.)
cuDNN	NVIDIA CUDA Deep Neural Network (GPU akcelerovaná knihovna funkcí pro implementaci hlubokých neuronových sítí)
DC	Direct Current (stejnosměrné napájení)
ELF	Executable and Linkable Format (standardní souborový formát pro uložení spustitelných souborů)
FLOPS	FLoating-point Operations Per second (počet operací v pohyblivé řádové čárce za sekundu)
GPU	Graphics Processing Unit (grafický procesor)
HDMI	High-Definition Multimedia Interface (nekomprimovaný obrazový a zvukový signál v digitálním formátu)
IoT	Internet of Things (Internet věcí)
IP	Internet Protocol (internetový protokol)
LAN	Local Area Network (lokální síť)
LED	Light-Emitting Diode (elektroluminiscenční dioda)
MEX	MATLAB EXecutable (soubory spustitelné MATLABem)
ONNX	Open Neural Network Exchange (otevřený formát pro výměnu neuronových sítí mezi nástroji)
PC	Personal Computer
ReLU	Rectified Linear Unit (usměrněná lineární aktivační funkce)
SD	Secure Digital (typ paměťových karet v přenosných zařízeních)
SSH	Secure Shell

SDK	Software Development Kit (sada vývojových nástrojů pro vývoj aplikací)
TPU	Tensor processing unit
UHS	Ultra High Speed (ultra vysoká rychlost – typ SD karet)
USB	Universal Serial Bus (univerzální sériová sběrnice)

Symboly:

U	napětí	[V]
P	výkon	[W]

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

V dnešní době se umělá inteligence dostala do aplikací pro běžné PC a postupně se začíná posouvat i do vestavných systémů. Tento posun je podpořen hardwarem, který má nízkou spotřebu a má dostatečný výpočetní výkon, většinou představovaný paralelními výpočetními jednotkami. Vedle řešení postavených na GPU, kde je hlavním představitelem firma nVIDIA, vznikají nová řešení. Jedním z nich je Tensor Processing Units (TPUs).

Vestavným systémem, o kterém tato práce pojednává, je jednodeskový počítač Jetson Nano od firmy NVIDIA. Který pro akceleraci výpočtů využívá GPU. Jsou zde představeny jeho základní vlastnosti, technické specifikace a postup, jakým nainstalovat software potřebný pro jeho funkčnost.

V této bakalářské práci je popsán současný způsob, jakým se umělá inteligence používá ve vestavných systémech. Je zde vysvětlen pojem At the edge, tedy způsob její implementace, který se začíná čím dál více rozšiřovat, a jeho výhody a omezení, které přináší.

Cílem této práce je vytvořit, naučit a následně implementovat v zařízení Jetson Nano neuronovou síť, která bude schopna rozpoznávat poruchu synchronního elektromotoru na základě předzpracovaných dat z měření na tomto motoru. Ta by měla v budoucnu sloužit pro vyhodnocení poruchy elektromotoru v reálném čase na základě dat získávaných z mikrokontroléru měniče. Pro složitost a množství možných poruch se práce omezila pouze na poruchu typu mezizávitový zkrat.

V této práci je napsán popis a představení základních vlastností nástrojů pro vývoj umělé inteligence. Nástroje představené v této práci, jsou podle zadání Keras, Tensorflow a Deep Learning Toolbox. Jako programovací jazyk pro práci se dodanými soubory naměřených a předzpracovaných dat byl zvolen programovací jazyk Python. Byl zvolen z důvodu jeho snadného použití a široké možnosti využití v oblasti umělé inteligence. S Deep Learning Toolbox-em se pracuje v prostředí programovacího jazyka MATLAB, který je používán v oblasti řízení.

Z uvedených nástrojů je v dalších částech práce použit Keras pro vytvoření a natrénování umělých neuronových sítí. Ty jsou následně optimalizovány a implementovány do vestavného systému Jetson Nano. Deep Learning Toolbox je využit při práci s naučeným Keras modelem neuronové sítě během implementace do zařízení Jetson Nano pomocí aplikace *GPU Coder*. U všech natrénovaných modelů neuronových sítí je změřena přesnost jejich klasifikace na validačních datech a doba potřebná pro klasifikaci vstupních dat.

Dosažené výsledky jsou porovnány v závěru práce.

1. UMĚLÁ INTELIGENCE VE VESTAVNÝCH SYSTÉMECH

V této kapitole jsou praktickými aplikacemi myšleny aplikace, které vyžadují využití vestavných systémů. Tyto systémy jsou většinou jednodeskové počítače, které jsou zabudovány do zařízení, které mají řídit.

1.1 At the edge

V současnosti se pro nasazení umělé inteligence do praktických aplikací většinou využívá cloudových služeb, na které jsou vestavné systémy napojeny. Vestavný systém použitý v zařízení tedy nemusí disponovat velkým výpočetním výkonem. Všechna data, která je potřeba zpracovat s využitím umělé inteligence, a tedy i velkého výpočetního výkonu jsou odesílána na cloud. Teprve na cloudu dochází k zapojení umělé inteligence do tohoto procesu a výstup z ní je odeslán zpět do vestavného systému, kde je zpracován.To je užitečné řešení pro aplikace, které jsou výpočetně velmi náročné, nevyžadují krátkou dobu odezvy, anebo u nich není možné tyto výpočty provádět na místě.

Čím dál více se však stává populární nasazení umělé inteligence přímo ve vestavných systémech. Tento přístup se nazývá “At the edge”, někdy také “On the edge”.

Hlavní myšlenkou At the edge je přesun výpočtů umělé inteligence z cloudu do vestavných systémů. Nejedná se o stěhování veškeré umělé inteligence z cloudu, ale především o výpočty, které jsou proveditelné ve vestavných systémech. Tomuto napomáhá velký rozvoj grafických čipů, které poskytují velký výpočetní výkon vzhledem k jejich malé spotřebě elektrické energie. Cloud bude využíván pro trénink umělé inteligence na obrovském množství dat. Bude spolupracovat s vestavnými systémy, které mohou během svého provozu sbírat data, a v době kdy nebudou plně využity, je odesílat na cloud. Díky těmto datům může být umělá inteligence v cloudu zdokonalována a následně aktualizována do vestavných systémů.[1]

Implementace umělé inteligence přímo v *edge* zařízeních (koncových zařízeních) přináší mnoho výhod.

Hlavní z nich jsou: [1][2][3]

- **Soukromí a bezpečnost** – Data mohou být zpracována přímo ve vestavném systému. Může tak odpadnout nutnost odesílání citlivých dat na cloud a snižuje se tak riziko jejich zneužití. Pro některé aplikace je využití umělé inteligence přímo v zařízení nezbytné. Jedná se například o průmyslové aplikace, bezpečnostní systémy, nebo provoz autonomních automobilů.

V takových případech musí být systém schopen vyhodnocovat data a reagovat v reálném čase.[1] U autonomních vozidel je navíc důležitá spolehlivá funkčnost i bez dostupného internetového připojení.

- **Doba odezvy** – Při využívání cloudových služeb je velkým problémem doba odezvy. Pokud jsou všechny výpočty provedeny přímo ve vestavném systému, odezva se může výrazně zkrátit, a to i mnohonásobně.

- **Snížení nákladů na přenos dat**– Nasazení umělé inteligence přímo v koncových zařízeních, snižuje nároky na potřebnou přenosovou kapacitu sítě, přes kterou je zařízení připojeno k serveru. A také může snížit náklady, pokud je připojení k internetu účtované podle objemu přenesených dat, nebo je nastaven datový limit (např. připojení IoT zařízení přes mobilní data).

1.2 Kvantizace modelu

Většina modelů neuronových sítí má své parametry, jako jsou váhy a prahy, uloženy v datovém typu float32.(TensorFlow, Keras, Deep Learning Toolbox). Takové modely jsou vhodné pro cloud nebo výkonný stolní počítač, avšak jejich použití ve vestavných systémech může být problematické. To hlavně z důvodu výpočetní náročnosti neuronové sítě, která se negativně promítne do výsledné rychlosti výpočtů a omezené velikosti operační paměti vestavného systému.

Z tohoto důvodu se využívá kvantizace modelu. Jedná se o optimalizační metodu, která převádí datové typy jeho parametrů na datové typy s menší přesností. Do kvantizace by měl vstupovat již natrénovaný model neuronové sítě. Výsledkem kvantizace je nový model neuronové sítě, který zabírá méně místa v paměti a je výpočetně méně náročný. Cenou za úsporu paměti a vyšší rychlost může být zhoršení přesnosti neuronové sítě.

Parametry modelu typu float32 jsou nejčastěji převáděny na float16, nebo int8.

Při kvantizaci na float16 bude mít model přibližně poloviční velikost. Při použití osmibitové kvantizace je možné dosáhnout čtvrtinové velikosti modelu a zvýšení rychlosti výpočtů i na více než trojnásobek.[4]

2. NVIDIA JETSON

NVIDIA Jetson Nano je jednodeskový počítač ze série Jetson, který je vyvíjen společností NVIDIA Corporation. Tyto počítače jsou určeny pro použití jakožto vestavné systémy v nichž může běžet umělá inteligence, nebo jiné algoritmy vyžadující vysokou úroveň paralelizace. Jejich předností jsou grafické čipy, které poskytují relativně vysoký výpočetní výkon vzhledem k jejich malé spotřebě elektrické energie. Celá série Jetson nabízí 4 modely, a to: *Jetson Nano, Jetson TX2 Series, Jetson Xavier NX* a *Jetson AGX Xavier.*[1] Některé z nich jsou dostupné ve více verzích a konfiguracích. Všechny modely Jetson využívají operační systém *Linux*, který je součástí instalačního souboru.

Obrázek 2.1 NVIDIA Jetson Nano [5]

2.1 NVIDIA Jetson Nano

Model Nano je s jeho rozměry 70 × 45 mm nejmenší a nejlevnější ze série Jetson. Jeho grafický čip je schopen provádět až 472 GFLOPs (472 miliard operací v pohyblivé řádové čárce za sekundu), a to při spotřebě pouhých 10 W. Z důvodu takto nízkého výkonu je zařízení vybaveno pouze pasivním chladičem, a je tedy schopné naprosto tichého provozu. Jetson Nano je možné používat ve dvou režimech napájení, a to režim se spotřebou 5 W, nebo 10 W. Při napájení přes micro USB konektor je doporučené využívat pouze 5W režim. Pro využití plného výkonu počítače je vhodné použít 5V stejnosměrný napájecí zdroj připojený na konektor *„DC Barrel jack“*. Deska modelu Nano je vybavena čtyřmi otvory, které umožňují jeho montáž do zařízení, které má řídit. Jako úložiště je v zařízení Jetson Nano použita paměťová karta.[6]

[1] Seřazeno vzestupně podle výpočetního výkonu.

TECHNICAL SPECIFICATIONS

GPU	NVIDIA Maxwell™ architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM® Cortex®-A57 MPCore processor
Memory	4 GB 64-bit LPDDR4
Storage	16 GB eMMC 5.1 Flash
Video Encode	4K @ 30 (H.264/H.265)
Video Decode	4K @ 60 (H.264/H.265)
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2 DPHY 1.1 (1.5 Gbps)
Connectivity	Gigabit Ethernet
Display	HDMI 2.0 or DP1.2 | eDP 1.4 | DSI (1 x2) 2 simultaneous
UPHY	1 x1/2/4 PCIE, 1x USB 3.0, 3x USB 2.0
I/O	1x SDIO / 2x SPI / 6x I2C / 2x I2S / GPIOs
Size	69.6 mm x 45 mm
Mechanical	260-pin edge connector

Obrázek 2.2 Technická specifikace NVIDIA Jetson Nano [6]

2.2 JetPack SDK

JetPack SDK je softwarová výbava pro zařízení z řady Jetson. Obsahuje balíček ovladačů pro Linux[2], operační systém Linux, knihovny využívající *CUDA*[3] (*TensorRT* a *cuDNN*) a APIs[4] pro podporu hlubokého učení, počítačového vidění a akcelerovaných výpočtů. Jeho instalací se zabývá kapitola 4.1. [7]

[2] *Linux Driver Package (L4T)*
[3] CUDA (Compute Unified Device Architecture) je softwarová a hardwarová architektura, která umožňuje urychlit výpočty, které lze provádět paralelně, pomocí grafických karet od firmy NVIDIA.[8]
[4] API (Application Programming Interface) je rozhraní pro programování aplikací. Toto rozhraní umožňuje komunikaci mezi dvěma aplikacemi.[9]

3. NÁSTROJE PRO VÝVOJ UMĚLÉ INTELIGENCE

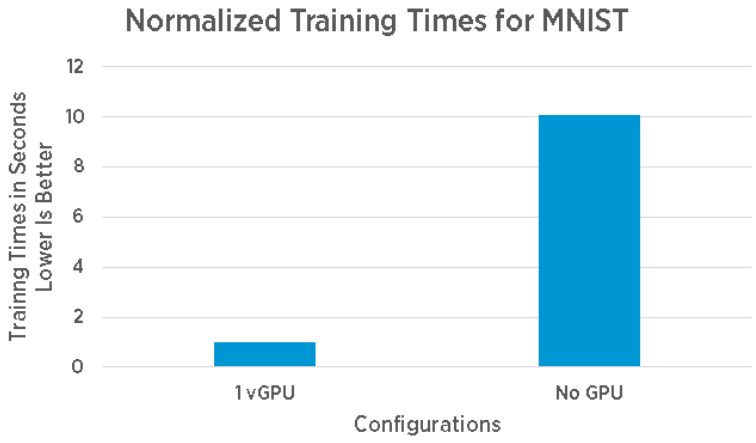
V této práci jsou pro vývoj umělé inteligence a její implementaci použity programovací jazyky *Python* a *MATLAB*, které umožnují použití nástrojů popsaných v následujících podkapitolách.

3.1 TensorFlow

TensorFlow je sada nástrojů určená pro vývoj umělé inteligence. Je vyvíjena společností Google a patří mezi nejpopulárnější nástroje používané pro její vývoj. Je k dispozici zdarma ke stažení s otevřeným kódem. Je tedy možné přečíst si všechny jeho zdrojové kódy a zdarma ho využívat i pro komerční použití pod licencí *Apache License 2.0*.

TensorFlow má dostupné API pro vývoj v programovacích jazycích *Python*, *JavaScript*, *C++*, *Java*, *Go* a *Swift*. Pro *Python* je API v současnosti nejkompletnější a nejjednodušší pro použití. API je rozhraní, které umožňuje využití funkcionalit TensorFlow za použití některého z uvedených programovacích jazyků, pro které jsou k dispozici knihovny. Výpočty, které tyto knihovny provádějí jsou následně pomocí API provedeny v jazyce C++ kvůli jeho vysoké rychlosti.

Při běžném použití využívá TensorFlow pro výpočty CPU (procesor) počítače. Učení a používání neuronových sítí je při použití samotného CPU poměrně pomalé. Pro zvýšení rychlosti je možné využít skrze TensorFlow platformu CUDA (spolu s ní je možné použití i cuDNN) od firmy NVIDIA Corporation. To umožňuje provádění výpočtů, které lze počítat paralelně, pomocí GPU (grafických jader). U neuronových sítí se provádí velké množství maticových výpočtů, které umožňují paralelní zpracování. Díky odlišné architektuře GPU oproti CPU, jsou schopny provádět paralelní výpočty mnohem rychleji. To znamená možnost mnohonásobného zkrácení doby potřebné k naučení neuronové sítě. Srovnání doby potřebné k naučení s využitím GPU a bez jejího využití je zobrazeno na obrázku 3.1.[10]



Obrázek 3.1 Srovnání doby učení s využitím GPU a bez využití GPU [11]

TensorFlow je kompatibilní pouze s grafickými kartami od firmy NVIDIA s architekturou CUDA, jejichž parametr *compute capability* je 3.5, 3.7, 5.2, 6.0, 6.1, 7.0, anebo vyšší než 7.0.[5]

3.2 Keras

Keras je API pro hluboké učení zaměřené na jednoduché použití uživateli v jazyce Python. Je postaven na sadě nástrojů TensorFlow, jehož použití zjednodušuje a dělá ho uživatelsky více přívětivé. Kód napsaný v Pythonu za použití knihoven Keras je tedy pomocí API vykonáván v TensorFlow (viz kapitola 3.1).

Na obrázku 3.2 je uvedena ukázka kódu v jazyce Python využívající knihovny Keras. V tomto kódu je vytvořena jednoduchá neuronová síť se dvěma skrytými vrstvami. Vstupem této sítě je dvourozměrná matice o rozměrech 10×10. Tato neuronová síť klasifikuje do tří kategorií.

```
1   from keras.models import Sequential
2   from keras.layers import Dense, Flatten
3
4   model = Sequential()
5   model.add(Flatten(input_shape=(10, 10)))
6   model.add(Dense(100, activation='relu'))
7   model.add(Dense(50, activation='relu'))
8   model.add(Dense(3, activation='softmax'))
9
10  model.compile(optimizer='Adam',
11                loss='sparse_categorical_crossentropy',
12                metrics=['accuracy'])
```

Obrázek 3.2 Ukázka kódu pro vytvoření neuronové sítě v Keras

Na prvních dvou řádcích kódu je proveden import tříd *Sequential*, *Dense* a *Flatten* z knihovny Keras.

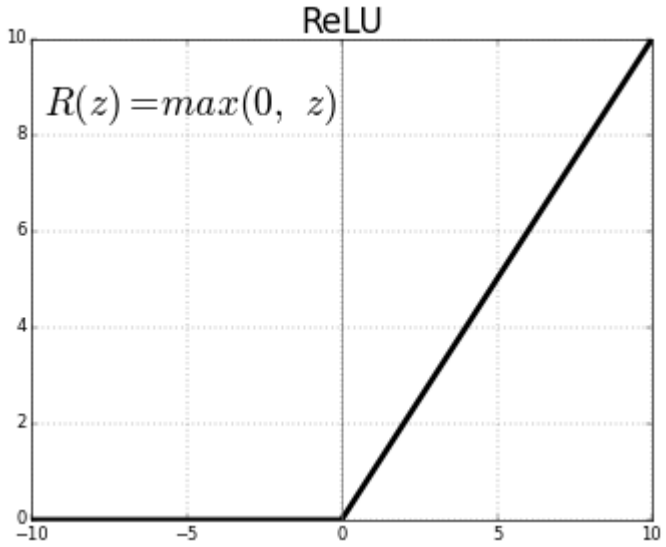
Sequential je typ modelu neuronové sítě, který má jeden vstupní a jeden výstupní tenzor. Na řádku číslo 4 se vytvoří instance třídy *Sequential*, která se uloží do proměnné *model*. Pomocí metody *add*, volané na řádcích 5 až 8, se do neuronové sítě přidávají jednotlivé vrstvy.[12]

Vrstva *Flatten* zajišťuje převod vstupního pole na vektor. V tomto případě převede rozměr vstupu z (10, 10), tedy dvourozměrné pole, na vektor o rozměrech (1, 100). *Dense* je plně propojená vrstva. Při jejím použití je potřeba zadat alespoň jeden vstupní parametr,

[5] Parametr *compute capability* lze zjistit podle modelu grafické karty na webové stránce: https://developer.nvidia.com/cuda-gpus [27.12.2020]

a to počet neuronů v této vrstvě. Na obrázku jsou tedy počty neuronů v jednotlivých skrytých vrstvách 100, 50 a 3 ve výstupní vrstvě.[13]

Na obrázku 3.2 je použit další vstupní parametr třídy *Dense*, a to aktivační funkce typu *ReLU* (*Rectified Linear Unit*). Pokud by nebyla zvolena aktivační funkce, Keras by automaticky použil výchozí aktivační funkci. Tou je pro *Dense* lineární aktivační funkce. *ReLU* (viz obrázek 3.3) se liší od lineární aktivační funkce tím, že má pro záporné vstupní hodnoty na výstupu hodnotu nula. Aktivační funkce *Softmax* je používána jako výstupní vrstva při řešení klasifikačních úloh. Tato funkce převádí vstupní hodnoty neuronu na pravděpodobnost, s jakou by síť klasifikovala danou kategorii jako správný výstup. Výstupem takové sítě je vektor pravděpodobností jednotlivých kategorií. Celkový součet všech těchto pravděpodobností ve výstupním vektoru je roven jedné. Určení výsledné kategorie se provede výběrem kategorie s nejvyšší pravděpodobností.[14]



Obrázek 3.3 Aktivační funkce *ReLU* [15]

Pomocí metody *compile* třídy *Sequential* se nastavují způsoby optimalizace modelu při tréninku a způsoby výpočtu chyby sítě. Keras nabízí použití celkem osmi optimalizačních metod, a to: *SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, Nadam, Ftrl.*[16]

Adam je výpočetně efektivní algoritmus, využívající *stochastic gradient descent*. Na rozdíl od klasické metody využívající *stochastic gradient descent*, která využívá jeden koeficient učení (označovaný jako alfa), používá *Adam* koeficienty učení zvlášť pro jednotlivé parametry sítě. Ty jsou navíc během procesu učení upravovány. *Adam* je vhodný pro použití v neuronových sítích s velkým počtem parametrů.[17]

SparseCategoricalCrossentropy je metoda výpočtu chyby sítě. Je vhodná pro použití v případech, kdy jsou na výstupu dvě a více kategorií. Tyto kategorie musí být

ve vzorových datech v celočíselném tvaru (int). Pokud by byly kategorie uvedeny jako binární *kód 1 z n*, je potřeba použít funkci *CategoricalCrossentropy*. V případě klasifikace do dvou kategorií je také možné použít funkci *BinaryCrossentropy*.[18]

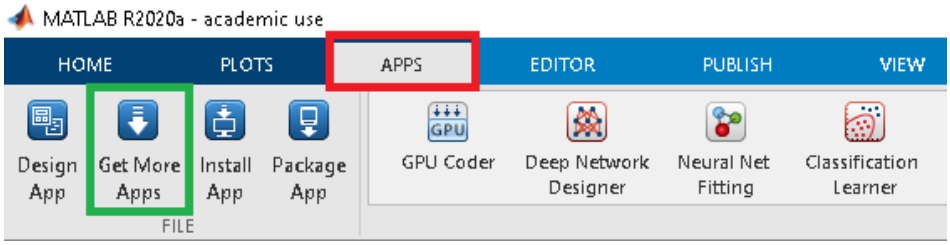
Metoda pro měření přesnosti sítě *Accuracy* podává jako výsledek poměr správně klasifikovaných vstupních vzorů, které jsou neuronové síti předkládány ku celkovému počtu předkládaných vzorů. Při procesu učení je tedy hlavním cílem tuto hodnotu přiblížit co nejvíce hodnotě rovné 1. Po vynásobení číslem 100 z této hodnoty dostaneme přesnost neuronové sítě v procentech.[19]

3.3 Deep learning Toolbox – MATLAB

Deep Learning Toolbox je sada nástrojů pro vývoj a implementaci umělé inteligence v prostředí MATLAB.

3.3.1 Instalace doplňků v MATLAB-u

Všechny doplňky (*Add-Ons*) pro MATLAB použité v této práci je možné nainstalovat pomocí nástroje *Add-On Explorer*. Ten je možné v MATLAB-u otevřít kliknutím na tlačítko „*Get More Apps*“ v záložce *APPS*. (Vyznačeno červeným a zeleným rámečkem na obrázku 3.4.)



Obrázek 3.4 Označení záložky a tlačítka pro otevření Add-On Exploreru v MATLAB-u

3.3.2 Deep Learning Toolbox

Pro používání sady nástrojů Deep Learning Toolbox je potřeba ji nainstalovat v MATLAB-u jako doplněk.

Je v něm možné poměrně jednoduchým a efektivním způsobem vytvářet a učit umělé neuronové sítě. Nástroje, které poskytuje Deep Learning Toolbox, umožňují řešit regresi, klasifikační úlohy, shlukování, redukci dimenzionality či předpovídání časových řad. Pro zpracování obrazu nabízí Deep Learning Toolbox dvě vstupní vrstvy a několik funkcí pro konvoluci, s nimiž je možné vytvořit konvoluční neuronové sítě. Stejně tak je možné použití již předučených neuronových sítí, kterých v současnosti Deep Learning Toolbox obsahuje 19. Například *squeezenet*, *googlenet*, *alexnet*, *resnet50* a další.

Deep Learning Toolbox umožňuje snadnou práci s modely neuronových sítí vytvořenými i v jiných nástrojích. Je v něm velice snadné používat modely neuronových sítí vytvořené v Keras nebo Caffe[6]. Ty je možné načíst pomocí funkcí *importKerasNetwork*[7] a *importCaffeNetwork*[8].

Dále je možný import a export modelů neuronových sítí ve formátu ONNX (Open Neural Network Exchange). Tento formát vznikl za účelem usnadnění spolupráce a převodu modelů neuronových sítí mezi již existujícími nástroji pro vývoj umělé inteligence, které jsou mezi sebou nekompatibilní. Práci s tímto formátem souborů umožňují v MATLAB-u funkce *importONNXNetwork* a *exportONNXNetwork*, které jsou součástí doplňku *Deep Learning Toolbox Converter for ONNX Model Format* pro MATLAB.[20][21]

Na obrázku 3.5 je ukázka kódu v MATLAB-u, kde na řádcích 10 až 22 je samotné vytvoření neuronové sítě. Řádky 1 až 8 slouží jen jako zjednodušená tréninková data pro tento příklad. Pro jednoduchost příkladu jsou tréninkovými daty pouze matice o rozměru 3×3 naplněné hodnotami nula, nebo jedna.

Na řádcích 11 až 16 jsou do matice *layers* přidávány jednotlivé vrstvy neuronové sítě. První vrstva je *imageInputLayer*. Jejími parametry jsou rozměr vstupního obrázku a název. První parametr vrstvy *fullyConnectedLayer* je počet neuronů. Druhým parametrem je název vrstvy. Na konci je přidána *classificationLayer*, která zajišťuje klasifikaci do jednotlivých kategorií na základě výstupního vektoru vrstvy softmaxLayer.

Na řádcích 18 až 20 je možné pomocí třídy *trainingOptions* upravovat parametry učení jako jsou například *Epsilon*, *InitialLearnRate*, *LearnRateDropFactor*, *MiniBatchSize*, *MaxEpochs*, *CheckpointPath* nebo *Plots*. Jediným povinným parametrem je optimalizační metoda, v uvedeném příkladu je to *adam*.

Třída *trainNetwork* použitá na řádku 22 zajistí trénink neuronové sítě, kterou jako instanci třídy nahraje do proměnné *neuronova_sit*. Jejími vstupními parametry jsou tréninková data *x_train* spolu s jejich výsledky *y_train*, vrstvy *layers* a parametry učení *options*.

[6] Caffe je sada nástrojů určená pro vývoj umělé inteligence. Je to projekt s otevřeným kódem vydaný pod licencí BSD.

[7] Součást doplňku *Deep Learning Toolbox Importer for TensorFlow-Keras Models*, který je pro použití této funkce nutné nainstalovat do MATLAB-u.

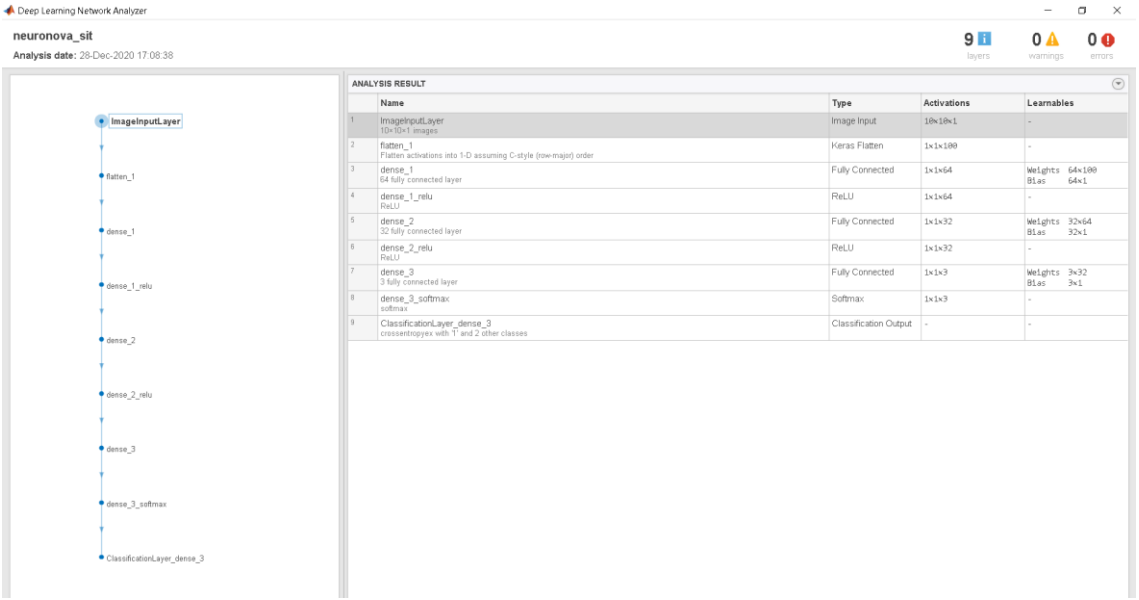
8 Součást doplňku *Deep Learning Toolbox Importer for Caffe Models*, který je pro použití této funkce nutné nainstalovat do MATLAB-u.

Průběh učení je možné sledovat v reálném čase pomocí grafického zobrazení, které je možné aktivovat změnou hodnoty parametru *Plots* na hodnotu *training-progress* ve třídě *options* (viz řádek 20 na obrázku 3.5).

```
 1 -    matice_nul = zeros(3, 3, 1, 2);
 2 -    matice_jednicek = ones(3, 3, 1, 2);
 3
 4 -    x_train = zeros(3, 3, 1, 2);
 5 -    x_train(:, :, :, 1) = matice_nul(:, :, :, 1);
 6 -    x_train(:, :, :, 2) = matice_jednicek(:, :, :, 1);
 7
 8 -    y_train = categorical([0; 1]);
 9
10 -    layers = [
11          imageInputLayer([3 3 1],"Name","imageInputLayer")
12          fullyConnectedLayer(9,"Name","fullyConnected_1")
13          fullyConnectedLayer(6,"Name","fullyConnected_2")
14          fullyConnectedLayer(2,"Name","fullyConnected_3")
15          softmaxLayer("Name","softmaxLayer")
16          classificationLayer("Name","classOutput")];
17
18 -    options = trainingOptions('adam', ...
19          'MiniBatchSize',2, ...
20          'Plots','training-progress');
21
22 -    neuronova_sit = trainNetwork(x_train,y_train,layers,options);
```

Obrázek 3.5 Ukázka kódu v MATLAB-u pro vytvoření neuronové sítě pomocí Deep Learning Toolbox

Již vytvořené neuronové sítě je možné vizualizovat pomocí funkce analyzeNetwork. Jak lze vidět na obrázku 3.6, tato funkce zobrazí v novém okně schéma vrstev sítě a tabulku s parametry těchto vrstev. Mezi parametry každé vrstvy je její název, typ aktivační funkce, rozměry a parametry, které se mění v průběhu učení.[22]



Obrázek 3.6 Ukázka vizualizace neuronové sítě pomocí funkce *analyzeNetwork*

4. IMPLEMENTACE UMĚLÉ INTELIGENCE NA ZAŘÍZENÍ NVIDIA JETSON NANO

Podkapitola 4.1 se věnuje spuštění a instalaci software na zařízení NVIDIA Jetson Nano. Další podkapitola je věnována postupu, na základě kterého je možné spustit na zařízení NVIDIA Jetson Nano programy využívající neuronové sítě napsané v programovacím jazyku Python (Keras a TensorFlow). Poslední podkapitola se věnuje implementaci neuronových sítí na zařízení NVIDIA Jetson Nano pomocí aplikace *GPU Coder* v prostředí MATLAB.

4.1 Instalace a zprovoznění NVIDIA Jetson Nano

Při instalaci operačního systému Linux na zařízení NVIDIA Jetson Nano bylo v této práci postupováno podle oficiálního návodu pro *Jetson Nano Developer Kit*[9].

Pro nastavení a první spuštění na Jetson Nano je nezbytné mít k dispozici paměťovou kartu microSD[10], klávesnici, myš a monitor, který se k Jetson Nano připojí pomocí konektoru *DisplayPort* nebo *HDMI*. Pro spuštění v takzvaném „headless mode" je možné místo monitoru použít druhý počítač. Řešení s použitím samostatného monitoru je však značně jednodušší.

Prvním krokem je volba napájecího adaptéru (viz kapitola 2.1). Pro použití 5V napájecího adaptéru připojeného konektorem DC Barrel Jack je potřeba zapojit zkratovací propojku (jumper) na konektor (pin) J48.

Na paměťovou kartu je potřeba nahrát soubor obrazu (*SD Card Image*) podle oficiálního návodu.[11]

Pro první spuštění je potřeba zapojit do Jetson Nano napájecí adaptér, klávesnici, myš, monitor (případně využít podle návodu postup bez monitoru) a paměťovou kartu (obsahující software podle návodu). Spuštění Jetson Nano je signalizováno zelenou LED, která se nachází vedle microUSB konektoru. Při prvním spuštění je uživatel proveden prvotním nastavením operačního systému Linux Ubuntu.[23]

[9] Návod je dostupný na webové stránce:
https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit [1.1.2021]
[10] Podle oficiálního návodu je doporučovaná třída paměťové karty alespoň UHS-1 s kapacitou minimálně 32 GB.[23]
[11] Návod je dostupný na webové stránce:
https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#write [1.1.2021]

4.2 Implementace neuronových sítí vytvořených v Keras a TensorFlow

Aby bylo možné spouštět na Jetson Nano programy napsané v jazyce Python, využívající knihovny Keras a TensorFlow, je potřeba nainstalovat navíc některé systémové balíčky, které vyžaduje TensorFlow a Python. Tento postup je uveden v oficiálním návodu[12] od firmy NVIDIA.

Po úspěšné instalaci *JetPack SDK* stačí spustit několik příkazů v terminálu systému Linux Ubuntu. První dva příkazy zajistí instalaci systémových balíčků, které vyžaduje TensorFlow.

```
$ sudo apt-get update
$ sudo apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev zip libjpeg8-dev liblapack-dev libblas-dev gfortran
```

Další dva příkazy provedou instalaci a aktualizaci *pip3*[13].

```
$ sudo apt-get install python3-pip
$ sudo pip3 install -U pip testresources setuptools==49.6.0
```

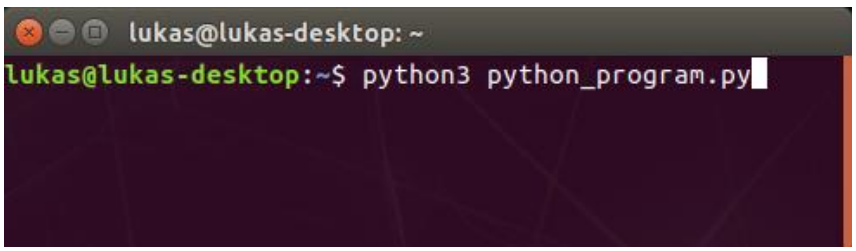
Následující příkaz slouží pro instalaci potřebných balíčků pro Python.

```
$ sudo pip3 install -U numpy==1.16.1 future==0.18.2 mock==3.0.5 h5py==2.10.0 keras_preprocessing==1.1.1 keras_applications==1.0.8 gast==0.2.2 futures protobuf pybind11
```

Poslední příkaz využívající *pip3* nainstaluje TensorFlow.

```
$ sudo pip3 install --pre --extra-index-url https://developer.download.nvidia.com/compute/redist/jp/v44 tensorflow
```

Pokud jsou všechny tyto příkazy úspěšně provedeny, pak je možné na zařízení Jetson Nano spouštět programy v jazyce Python, využívající knihovny Keras a TensorFlow.[24] Tyto programy lze spouštět například v terminálu příkazem *python3 název_souboru.py*, jako na obrázku 4.1.



Obrázek 4.1 Spuštění programu v jazyce Python v terminálu zařízení NVIDIA Jetson Nano

[12] Návod je dostupný na webové stránce: https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html#overview

[13] Pip (pip3) je správce balíčků jazyka Python. Pomocí vlastních příkazů umožňuje práci s nimi.

4.3 Implementace neuronových sítí vytvořených v prostředí MATLAB

MATLAB není možné nainstalovat na zařízení NVIDIA Jetson Nano. Neexistuje totiž verze MATLAB-u pro ARM procesory. Spuštění samotného MATLAB kódu přímo v zařízení Jetson Nano tedy není možné.

Pro spuštění programu, vytvořeného v MATLAB-u, na NVIDIA Jetson je potřeba využít aplikaci *GPU Coder*. Ta se musí nainstalovat jako doplněk do MATLAB-u. Z MATLAB kódu je pomocí aplikace *GPU Coder* vygenerován vysoce optimalizovaný kód *CUDA*, který dokáže akcelerovat výpočty pomocí grafického čipu. K tomu využívá především knihovny *cuDNN* a *TensorRT*. Tento kód může být zkompilován jako zdrojový kód, statická knihovna, dynamická knihovna, nebo soubor MEX pro spuštění v MATLAB-u. Kód může být zkompilovaný pro stolní počítače, servery, nebo pro vestavné zařízení z řady *NVIDIA Jetson* i *NVIDIA DRIVE*.[25]

Použití aplikace GPU Coder pro generování *CUDA* kódu pro zařízení NVIDIA Jetson vyžaduje mít v počítači nainstalovaný program *Microsoft Visual Studio*. Při generování kódu je totiž využíván jeho C++ kompilátor. Dále je potřeba nainstalovat *CUDA*[14], *cuDNN*[15] a *TensorRT*[16] od firmy *NVIDIA Corporation*.

Před použitím aplikace *GPU Coder* je nutné v MATLAB-u nastavit cestu k prostředí cuDNN a TensorRT. Dále je musí nastavit C a C++ kompilátor. K tomu je možné využít příkazy[17]:

```
setenv('NVIDIA_CUDNN','C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v11.2')
setenv('NVIDIA_TENSORRT','C:\TensorRT-5.1.5.0')
mex -setup
mex -setup c++
```

Poslední dva příkazy nastaví C a C++ kompilátor, které budou používány. Pokud je na počítači nainstalované *Visual Studio*, tyto příkazy nastaví kompilátory, které jsou dodávány s programem *Visual Studio*. Pro použití jiného kompilátoru stačí příkazy upravit na následující:

```
mex -setup:cesta_ke_kompilátoru C
mex -setup:cesta_ke_kompilátoru C++
```

Dostupnost potřebných nástrojů, které jsou nezbytné pro vygenerování spustitelného souboru pro NVIDIA Jetson, je možné ověřit následujícím příkazem.

```
coder.checkGpuInstall('full')
```

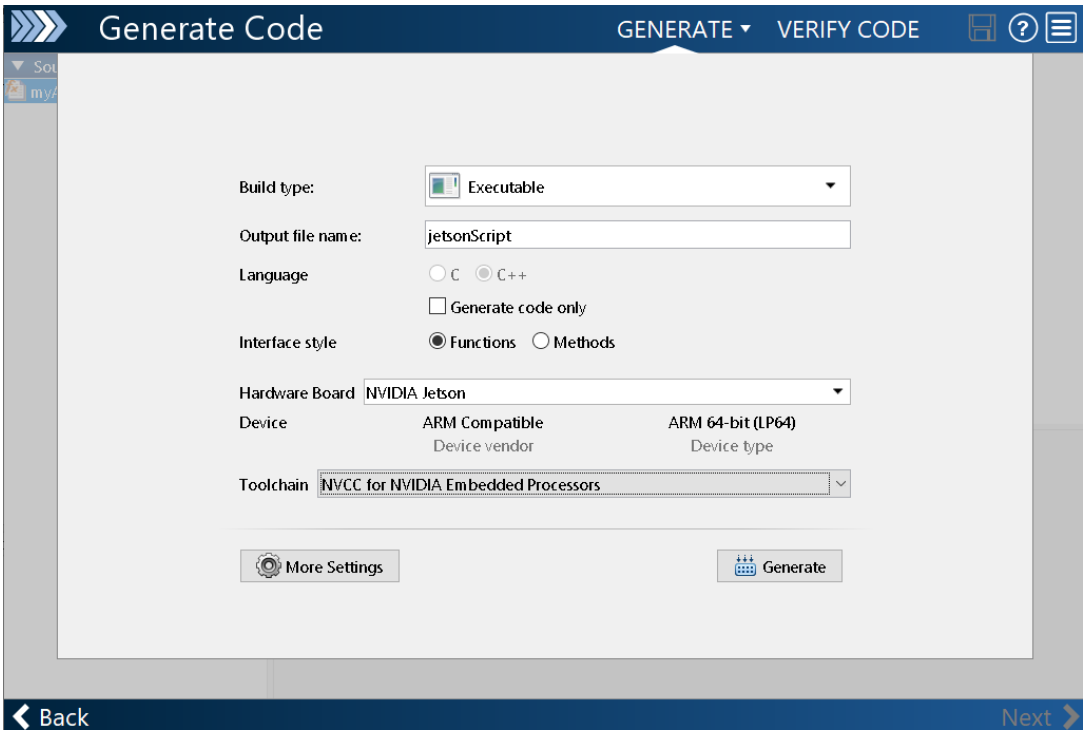
[14] Odkaz na oficiální návod k instalaci *CUDA*: https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

[15] Odkaz na oficiální návod k instalaci *cuDNN*: https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html

[16] Odkaz na oficiální návod k instalaci TensorRT: https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html

[17] Druhý parametr funkce *setenv* je cesta k adresáři, ve kterém jsou CUDA a TensorRT nainstalovány.

Aby bylo možné kód zkompilovat pro spuštění na NVIDIA Jetson je potřeba nejdřív nainstalovat do MATLAB-u doplněk *NVIDIA Jetson Support from MATLAB Coder*. Po instalaci se v aplikaci *GPU Coder* zpřístupní volba *NVIDIA Jetson* v nabídce pro výběr *Hardware Board* a volba *NVCC for NVIDIA Embedded Processors* v nabídce pro výběr *Toolchain.* (Zobrazeno na obrázku 4.2)

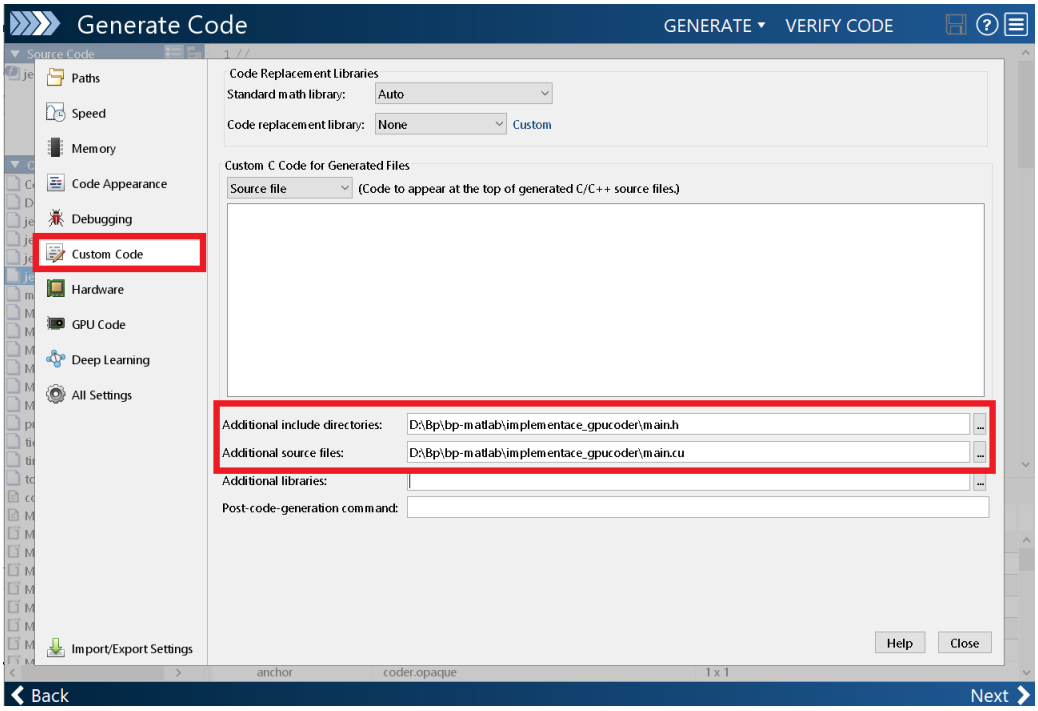


Obrázek 4.2 Ukázka aplikace GPU Coder

Pro vygenerování spustitelného souboru je nutné mít v MATLAB-u kód ve funkci. Po spuštění aplikace GPU *Coder* je uživatel proveden celým procesem, který vede od výběru funkce, ze které se má CUDA kód generovat až po samotné vygenerování kódu. Proces se skládá z pěti kroků. V prvním kroku se zvolí název funkce, ze které chce uživatel kód generovat. Ve druhém kroku se definují vstupní datové typy funkce. Ty je možné nadefinovat ručně, nebo automaticky. Pro automatické nadefinování vstupních datových typů je potřeba vytvořit script, který volá zadanou funkci. V aplikaci *GPU Coder* se zadá tento skript a vstupní datové typy se nadefinují automaticky. Ve třetím kroku se funkce testuje proti případným chybám, které by bylo obtížné odhalit ve vygenerovaném kódu. Ze zadané funkce si *GPU Coder* vytvoří *MEX* funkci, která se testuje na CPU a GPU. Odhalené chyby jsou vypsány. Pokud se test povede a neodhalí se chyby, přejde se do kroku zobrazeném na obrázku 4.2, odkud je možné generovat CUDA kód.

Aby byl vygenerovaný kód spustitelný na zařízení NVIDIA Jetson, je nezbytné před jeho vygenerováním nastavit *Build type* na *Executable*. To zajistí, že se vygenerovaný

kód zkompiluje do spustitelného souboru s příponou *elf*. Pro kompilaci do spustitelného souboru je potřeba přidat soubory *main.cu* a *main.h*. v nastavení podle obrázku 4.3. Soubory *main.cu* a *main.h* jsou aplikací *GPU Coder* vygenerovány v základní podobě. Funkce v nich je nutné upravit a tyto upravené soubory přidat.



Obrázek 4.3 Obrázek přidání souborů pro vygenerování spustitelného souboru v aplikaci *GPU Coder*

GPU Coder umožňuje vygenerované soubory přímo nahrát do zařízení NVIDIA Jetson pomocí SSH. Podmínkou je, aby byly obě zařízení připojeny do stejné LAN. Přístup do zařízení NVIDIA Jetson pomocí SSH umožňuje *JetPack SDK*. Přístup přes SSH se dá v *GPU Coder* jednoduše nastavit vyplněním přístupových údajů. V *More Settings* v záložce *Hardware* stačí vyplnit *Device Address* – tedy IP adresu zařízení, přihlašovací jméno a heslo, které se nastavilo pro uživatelský profil na operačním systému Ubuntu, a *Build Directory* – adresář, do kterého se má nahrát adresář s vygenerovanými soubory.

Zkompilovaný soubor s příponou *elf* je možné spustit v systému Ubuntu příkazem v terminálu.[18]

```
./jetsonScript.elf
```

4.3.1 Implementace natrénované neuronové sítě na předložená data

V přímé návaznosti na kapitolu 4.3 je v této podkapitole popsán postup, který byl použit

[18] Pokud je soubor pojmenován jinak, název „jetsonScript“ se nahradí.

pro implementaci natrénovaného Keras modelu pomocí aplikace *GPU Coder*.

Funkcí pro generování kódu je *jetsonScript*.

```
function output = jetsonScript(vzorek)

    persistent neuronova_sit
    if isempty(neuronova_sit)
        neuronova_sit=
              coder.loadDeepLearningNetwork("keras_model.mat");
    end

    tic;
    klasifikace = neuronova_sit.predict(vzorek);
    [~, argmax] = max(klasifikace);
    toc;
    output = [argmax toc];
end
```

Keras model musí být převeden do MATLAB Data souboru, protože v aplikaci *GPU Coder* není možné použít funkce pro načítání modelů neuronových sítí uložených v souboru, který používá Keras (přípona souboru *h5*) ani ONNX (přípona souboru *onnx*). Převod je realizován v MATLAB skriptu *prevod_keras_modelu.m*.

Funkce má jako vstupní argument vektor šestnácti hodnot, který je vstupem neuronové sítě. Dále je načtena neuronová síť za *MATLAB Data* souboru, které je vstupní vektor předložen ke klasifikaci. Proměnná argmax obsahuje index s nejvyšší hodnotou pravděpodobnosti výstupního vektoru vrstvy *Softmax*. Tímto způsobem je určena výsledná kategorie klasifikovaných dat. Výstupem funkce je vektor dvou hodnot. První hodnota je kategorie, do které byla vstupní data klasifikována a druhá hodnota je změřený čas, potřebný pro klasifikaci.

Pro testování této funkce v aplikaci *GPU Coder* byla vytvořena funkce jetsonScriptTest, která pouze volá funkci jetsonScript se vstupním argumentem.

Po vygenerování souborů z této funkce, byl vygenerovaný soubor *main.cu*, upraven a spolu s vygenerovaným souborem *main.h* vložen jako zdrojový soubor pro generování kódu tak, jak je zobrazeno na obrázku 4.3. [19]

Do funkce *main* v souboru *main.cu* byl dopsán kód, který zajišťuje načítání vstupních vzorků pro neuronovou síť z CSV souboru, který je uložen ve stejném adresáři jako spustitelný ELF soubor a musí mít název „*data.csv*“. Správná kategorie dat je uvedena na každém řádku CSV souboru. Dále tento kód načtená data předkládá neuronové síti ke klasifikaci a zaznamenává výstupy klasifikace všech vstupních dat z CSV souboru. Na základě těchto dat je spočítána celková přesnosti klasifikace na všech předložených datech a průměrná doba potřebná pro klasifikaci. Tyto dvě vypočtené hodnoty jsou vypsány do konzole na konci programu.

Zápis dat z matice *DQ_oscilacie_Filtrovanie16* do CSV souboru s přidáním sloupce, který obsahuje správnou kategorii dat, je realizován v MATLAB skriptu *data_to_csv.m*

[19] Soubory *main.cu* i *main.h* jsou k dispozici v příloze ve složce bp_matlab.

Implementace byla vyzkoušena na modelech číslo 1, 3 a 5 uvedených v tabulce 6.1. V tabulce 4.1 jsou uvedeny změřené doby potřebné pro klasifikaci vstupních dat těmito modely. Na vybraných souborech použitých jako vstup neuronové sítě byla pozorována přesnost klasifikace srovnatelná s původním Keras modelem.

Tabulka 4.1 Tabulka změřených časů potřebných pro klasifikaci vstupu modelu implementovaného na zařízení Jetson Nano pomocí aplikace *GPU Coder*

			doba klasifikace [µs]
číslo modelu	TIMESTEPS	počty neuronů ve vrstvách	GPU Coder (Keras model)
1	1	16-8-2	822
3	1	32-16-2	798
5	1	64-32-16-2	953

5. PŘEDLOŽENÁ DATA

Předložená data jsou vygenerována z dat získaných měřením na skutečném synchronním elektromotoru. Porucha se projevuje na druhé harmonické frekvenci měřených proudů a napětí. Pro práci s předloženými daty, vytvoření a natrénování neuronové sítě, byl zvolen programovací jazyk Python.

5.1 Zadaná data

V rámci zadání této práce byly dodány soubory obsahující data naměřená při pokusech se synchronním elektromotorem. Tato data jsou uložena v souborech typu *MATLAB Data*. Soubory jsou rozděleny názvem na ty, které obsahují data naměřená na motoru s poruchou a na soubory obsahující data naměřená na motoru bez poruchy. Dále jsou soubory označeny podle místa v motoru, kde porucha nastala a podle zátěže měřené na výstupu motoru pomocí dynamometru.

Spolu s daty byl dodán také MATLAB skript, který z naměřených dat vybírá vyfiltrovanou druhou harmonickou frekvenci a různé další příznaky. Tento skript také simuluje poruchu při zkratu 3 závitů vinutí fáze na všech fázích motoru na základě poruchy naměřené na jedné fázi motoru. Tato porucha se v datech vyskytuje vždy pouze na jedné fázi. V datech tedy nenastává situace, kdy by se vyskytovala porucha na více fázích současně.

V souborech, ze kterých se generují výše popsaná data, se nachází data z měření, které bylo prováděno s vzorkovací frekvencí 10 kHz. Délka většiny průběhů měřených veličin je 410 000 vzorků, což odpovídá běhu motoru po dobu 41 sekund. Tato data neobsahují prvních 70 000 vzorků, které byly pořízeny v době, kdy se při měření na motoru teprve zapínal dynamometr pro měření točivého momentu.

Spuštěním skriptu se vygeneruje celkem 384 souborů o celkové velikosti na disku přibližně 108 GB. Obsahem vygenerovaných souborů jsou matice, ve kterých jsou zaznamenány již předzpracované průběhy napětí a proudů v motoru a také jeho rychlost. Matice s názvem *DQ_oscilacie_Filtrovanie16* obsahuje vyfiltrovanou druhou harmonickou složku z měřených napětí a proudů v motoru. Tato skriptem vygenerovaná data je podle pokynů k použití skriptu možné použít přímo na vstupu neuronové sítě. A to bez nutnosti dalšího předzpracování. Soubory obsahují také matici datového typu *categorical* s názvem *Class*, ve které je pro každý vzorek uložena informace o přítomnosti poruchy. Matice obsahuje hodnoty *Normal, FaultSubSys1* a *FaultSubSys2*. Soubory jsou ovšem vždy jen s poruchou, anebo bez poruchy. Hodnoty matice *Class* jsou tedy pro všechny vzorky v souboru vždy stejné.

5.2 Zpracování předložených dat

Pro zpracování souborů *MATLAB Data.* byla v Python projektu vytvořena třída *MatlabData*, která zajišťuje práci s těmito soubory. Načítání *MATLAB Data* souborů zajišťuje funkce *loadmat* z knihovny *SciPy*.

Funkce *loadmat* vrací data ze souboru jako slovník, obsahující data z načteného souboru. Data, která jsou v MATLAB-u reprezentována jako matice, jsou v jazyce Python načtena ze souboru jako datový typ *numpy array*.

O zpracování dat od souborů až po tréninkovou a validační množinu se v Python kódu starají třídy *MatlabData, Dataset* a TrainingSet.

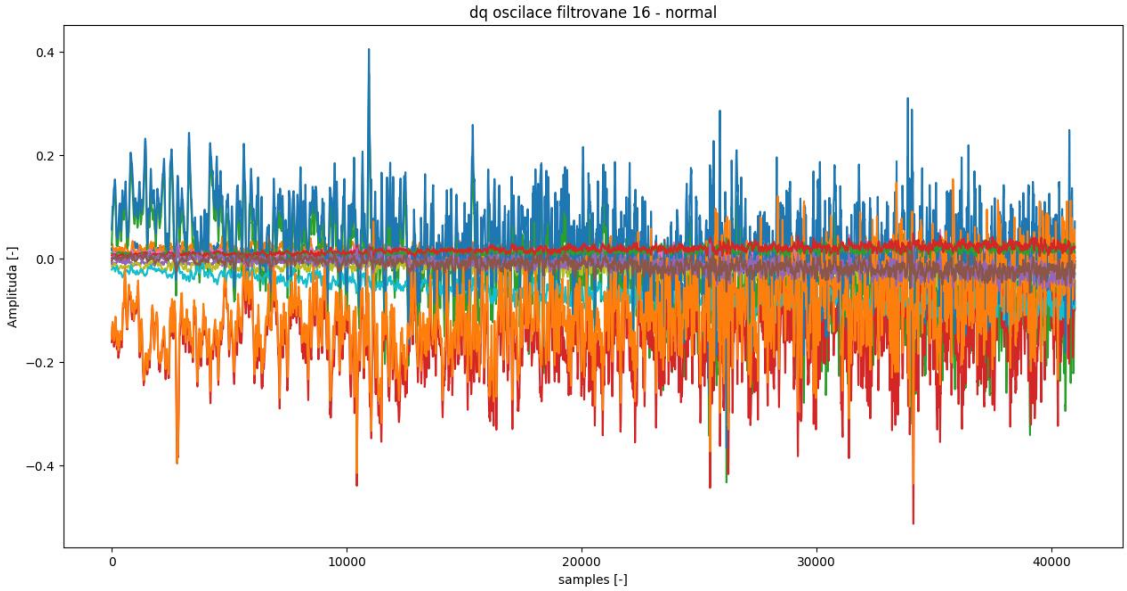
- *MatlabData* zprostředkovává načítání dat vždy z jednoho souboru *MATLAB Data.*
- Třída *Dataset* pracuje vždy s jednou instancí třídy *MatlabData*, ze které načítá data. Ta jsou následně promíchána, a to vždy ve stejném pořadí pro každý soubor. To z důvodu, aby tréninková a validační data, na která jsou následně data z každého souboru rozdělena, obsahovala vzorky, které byly naměřeny při různých rychlostech motoru. Nedojde tedy k tomu, že by například tréninková data obsahovala pouze vzorky z měření do určité rychlosti motoru a validační data zase vzorky pouze od určité rychlosti motoru.
- Metody třídy *TrainingSet* prochází seznam cest k *MATLAB Data* souborům, který je předán jejímu konstruktoru. Pro každý soubor vytvoří instanci třídy *Dataset*, a data z něj přidává do svých proměnných, které obsahují tréninková a validační data.

Třída *DataVisualizer* obsahuje metody *plot_proudy*, *plot_napeti*, *plot_oscilace* a další, které vykreslují veličiny z vygenerovaných souborů do grafů. Při pozorování těchto grafů, bylo zjištěno, že jsou si vykreslené průběhy každé veličiny s poruchou velmi podobné s těmi bez poruchy. Velký rozdíl se dá již pouhým pohledem na graf pozorovat u dat s názvem *DQ_oscilacie_Filtrovanie16*.[20] Typickým znakem pozorovatelným na grafech, které obsahují data s poruchou, je, že se průběhy veličin vzdalují od osy x. Zatímco na grafech, které jsou bez poruchy oscilují veličiny kolem hodnot, které jsou blízko osy x a nerozbíhají se. Na základě tohoto zjištění byla data *DQ_oscilacie_Filtrovanie16* vybrána jako ta, na kterých se bude neuronová síť učit rozpoznávat poruchu motoru.

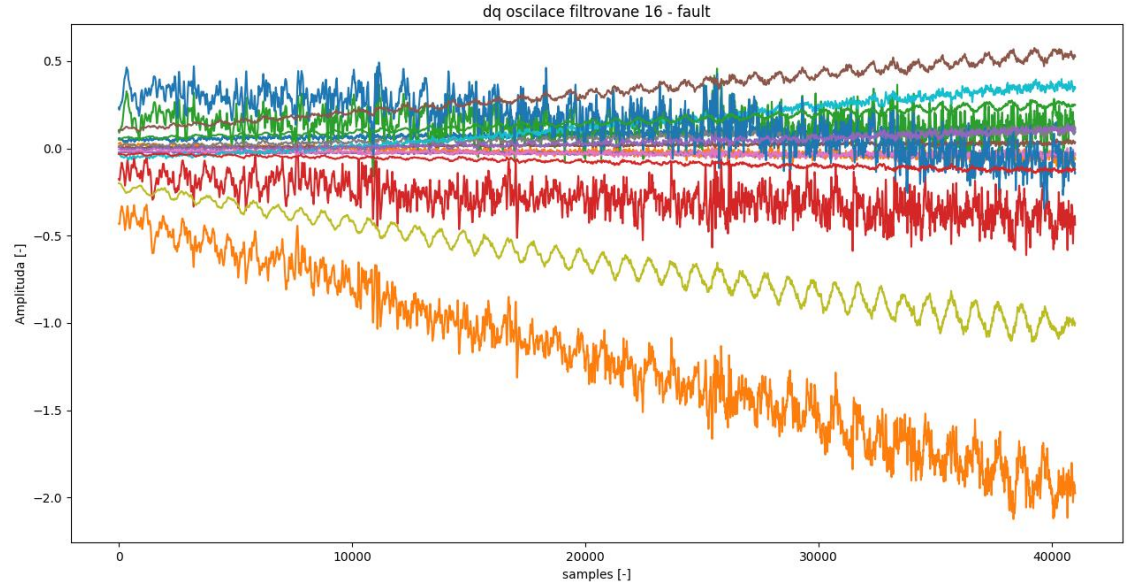
Funkce *loadmat* nenačítá z *MATLAB Data* souborů matici s názvem *Class*. Určení kategorie dat tedy řeší metoda *select_category* ve třídě *MatlabData*. Ta zjišťuje přítomnost slova „fault“ v názvu souboru po jeho načtení. Názvy souborů jsou vždy důkladně popsány a je podle nich zřejmé, jaká data soubor obsahuje V konstruktoru třídy se výstup *select*_category uloží do class variable (proměnná třídy) *FAULT*, která při hodnotě *True* znamená, že data obsahují poruchu. Neuronová síť bude schopna pouze

[20] Zobrazeno na obrázcích 5.1 a 5.2

detekovat přítomnost poruchy. Nebude tedy schopna určit, ve které části motoru porucha nastala. Data obsahující poruchu jsou dále ve vzorcích, které jsou síti předkládány pro učení a validaci označena číslem 1 (kategorie 1) a data bez poruchy číslem 0 (kategorie 0).



Obrázek 5.1 Obrázek grafu průběhů veličin z dat *DQ_oscilacie_Filtrovanie16* bez poruchy



Obrázek 5.2 Obrázek grafu průběhů veličin z dat *DQ_oscilacie_Filtrovanie16* s poruchou

6. NATRÉNOVÁNÍ UMĚLÉ NEURONOVÉ SÍTĚ NA PŘEDLOŽENÁ DATA

Cílem je natrénovat umělou neuronovou síť tak, aby byla schopna vyhodnotit z předložených dat, jestli se jedná o data z měření na motoru s poruchou, nebo bez poruchy.

6.1 Zpracování dat pro vstup umělé neuronové sítě

Jak bylo zmíněno v kapitole 5.2, pro učení neuronové sítě byla zvolena data, která jsou ve vygenerovaných souborech uloženy v matici s názvem *DQ_oscilacie_Filtrovanie16*.

Pro vytváření vstupních vzorků pro neuronovou síť byly v této práci vyzkoušeny dva přístupy. Oba umožňují zavedení paměti do vstupních dat.

Je potřeba zdůraznit, že použitá data nejsou ani při hodnotě *TIMESTEPS* rovno jedné zcela bez paměti. Protože byl na používaná data při jejich předzpracování (v rámci vygenerování souborů zadaným MATLAB skriptem) aplikován filtr, který pracuje s pamětí, tak jednotlivé vzorky v matici DQ_oscilacie_Filtrovanie16 paměť obsahují.

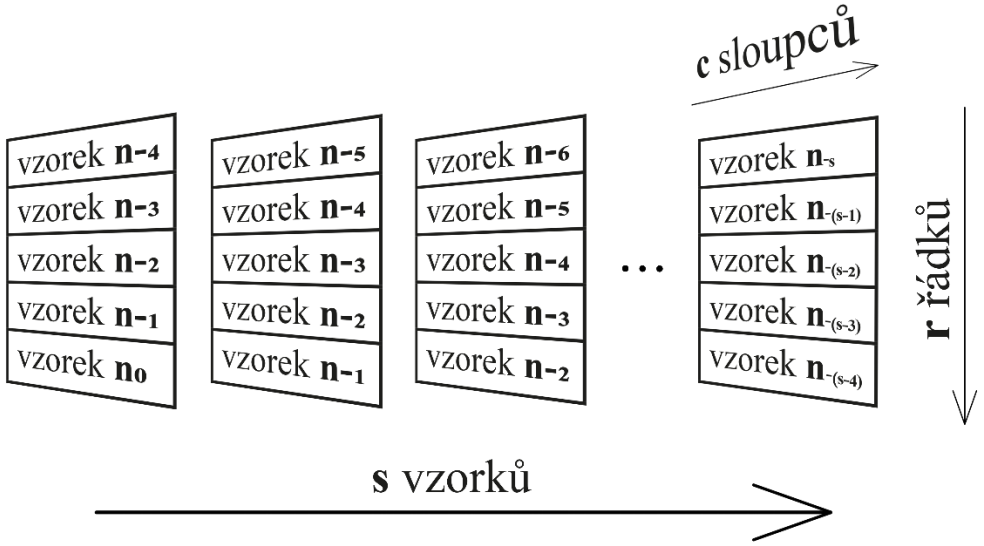
6.1.1 Vstupní data jako trojrozměrné pole

První přístup pro vytváření vstupních dat pro neuronovou síť, který byl zvolen, je využíván při práci s rekurentními neuronovými sítěmi. Spočívá v použití trojrozměrných polí místo dvourozměrných pro uložení tréninkových a validačních dat. Ty potom neobsahují pouze jeden vzorek z dat *DQ_oscilacie_Filtrovanie16*, ale sérii několika vzorků, které jdou v čase po sobě. Takže pro každý vzorek z původních dat má neuronová síť na vstupu aktuální vzorek a několik vzorků, které mu předcházejí. Výsledné trojrozměrné pole je tedy složeno z dvourozměrných polí, ve kterých jsou v řádcích vzorky jdoucí po sobě. Jednotlivé sloupce potom obsahují samotná data pro každý vzorek. To umožňuje dostat na vstup neuronové sítě data s pamětí, i když je používána dopředná neuronová síť bez paměti.

Toto pole se vzorky je přímo předkládáno neuronové síti při procesu učení, kdy do neuronové sítě vstupují dvourozměrné pole, které si neuronová síť pomocí vrstvy typu *Flatten* převádí na vektor hodnot. Použití této paměti ve vzorcích není nezbytné, její velikost se dá nastavit pomocí class variable *TIMESTEPS* třídy *Config*. Pro použití vstupních dat bez přidané paměti stačí *TIMESTEPS* nastavit na hodnotu 1.

Pole vstupních dat (*numpy array*) má výsledný rozměr ve tvaru (počet_vzorků, hloubka_paměti, počet_hodnot_ve_vzorku). *hloubka_paměti* představuje parametr *TIMESTEPS*. Toto uspořádání vzorků v poli vstupních dat je ilustrováno na obrázku

6.1[21]. Na obrázku jsou pro názornost vzorky zjednodušeny pouze na „buňku v tabulce“. Každý vzorek je v reálných datech vektorem šestnácti hodnot. Řádky představují *TIMESTEPS*. Osa *s vzorků* představuje výsledné vzorky vytvořené pro vstup neuronové sítě. V Pythonu jsou rozměry polí *numpy array* reprezentovány jako datový typ *tuple*. Obecně má pole z obrázku 6.1 rozměr (s, r, c), tedy počet vstupních vzorků neuronové sítě (s) a počet řádků (r) a sloupců (c) v každém vstupním vzorku. Takže pole tréninkových dat, které by bylo vytvořeno pouze z jednoho souboru by mělo rozměr (286993, 5, 16)[22] při nastavení *TIMESTEPS* na hodnotu 5.



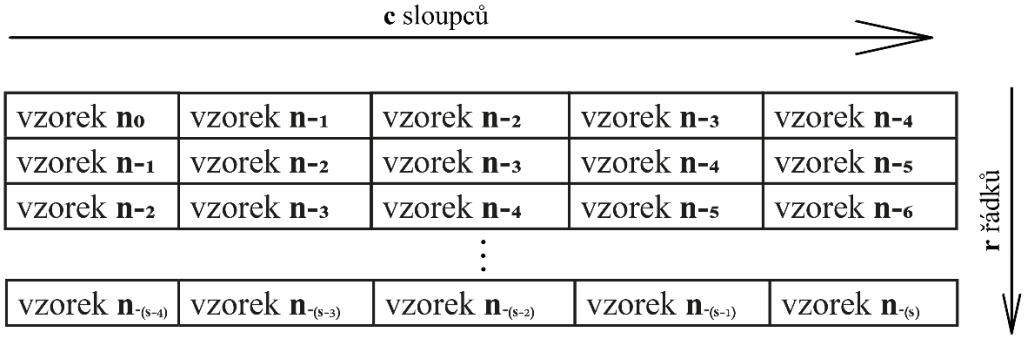
Obrázek 6.1 Ilustrace trojrozměrných vstupních dat pro umělou neuronovou síť

6.1.2 Vstupní data jako dvourozměrné pole

Druhý přístup, který byl použit, využívá pro uložení vstupních dat neuronové sítě dvourozměrné pole. Rozdíl je v tom, že se vzorky, které jsou použity jako paměť, přidávají do řádků. Není tedy potřeba třetího rozměru pole a zvýší se tak celkový počet sloupců v poli na hodnotu, odpovídající součinu hodnoty *TIMESTEPS* a počtu hodnot ve vzorku. Tento postup vytváření vstupních vzorků pro neuronovou síť ilustruje obrázek 6.2. Pole tréninkových dat, vytvořené z jednoho souboru, by mělo rozměr (286993, 80) (pro TIMESTEPS=5).

[21] Obrázek je pouze ilustrací. Zobrazuje příklad dat, kdy je hodnota *TIMESTEPS* rovna pěti. Pořadí dat je na něm zobrazeno jinak, než ve skutečnosti je, protože při zpracování vstupních dat je pořadí vzorků v ose *s* promícháno.

[22] Hodnota 286993 je přibližně 70 % ze 410000 vzorků uložených v matici DQ_oscilacie_Filtrovanie16.



Obrázek 6.2 Ilustrace dvourozměrných vstupních dat pro umělou neuronovou síť

6.1.3 Požadované výstupy neuronové sítě

Pro oba výše popsané přístupy vytváření vstupních dat pro neuronovou síť se používají stejná pole obsahující požadované výstupy neuronové sítě pro každý vstupní vzorek. Ve třídě *TrainingSet* jsou tréninková data uložena v proměnné *x_train*. Pro toto pole je vektorem požadovaných výstupů proměnná *y_train*. U validační dat je situace obdobná. V proměnné *x_test* jsou uloženy validační data a v *y_test* jejich správné výsledky. Proměnné *y_train* a *y_test* jsou, stejně jako *x_train* a *x_test*, datového typu *numpy array* o rozměru (počet_vzorků, 1), kde počet_vzorků odpovídá počtu vstupních vzorků v datech *x_train*, nebo *x_test*. Pole má tedy pouze jeden sloupec, který obsahuje hodnoty 0 nebo 1.

6.2 Rozdělení dat na tréninková a validační

Tréninková data jsou neuronové síti předkládána při procesu učení sítě. Validační data jsou ta, na kterých je prováděno vyhodnocení přesnosti sítě. Je důležité, aby se validační data neobjevila v tréninkových datech. Potom by vyhodnocení sítě nebylo provedeno jen na nezávislých datech a mohla by být změřena vyšší přesnost klasifikace, než jaké je neuronová síť ve skutečnosti schopna dosáhnout. V této práci je 70 % dat určeno pro učení a 30 % dat pro validaci.

Jak je popsáno v kapitole 5.2 v odstavci o třídě *Dataset*, tak právě tato třída zajišťuje rozdělení dat, a to pomocí metody *train_test_split*. Ta rozděluje seznam vzorků, které jsou ji předány argumentem funkce, na data tréninková a validační. Jelikož metoda pracuje se seznamy vzorků, které jsou již promíchány, stačí seznamy rozdělit indexem, který je vypočítán podle poměru, který nastavuje class variable *TEST_SAMPLES_RATIO* třídy *Config*. Ten pro výše uvedený poměr 70:30 odpovídá hodnotě 0.3.

Pokud by učení sítě neproběhlo na všech souborech, a byla spuštěna validace neuronové sítě, tak algoritmy, které byly pro tuto práci navrženy, umožňují použití všech dat ze souborů, na kterých se síť neučila. To z důvodu, aby bylo možné využít pro validaci více dat, a tím neuronovou síť lépe otestovat. Jelikož se na nich síť neučila, jsou tato data

stejně tak nezávislá pro validaci, jako běžná validační data ze souborů, které byly použity pro učení sítě.

6.3 Vytvoření umělé neuronové sítě

Pro vytvoření a práci s umělou neuronovou sítí byla použita sada nástrojů *Keras*. Neuronovou síť implementuje třída *NNModel*. V ní jsou metody pro vytvoření a načítání modelu neuronové sítě. Při vytvoření instance této třídy se v konstruktoru volá metoda *load_model*, která pomocí dalších metod kontroluje, zda již neexistuje soubor, obsahující model sítě. V případě, že model není na disku nalezen, je automaticky vytvořen nový. Třída také obsahuje metodu *predict*, které se argumentem předává vzorek dat, který se má klasifikovat. Metoda *create_model* vytváří neuronovou síť, jako *Sequential* model, který je složen ze vstupní vrstvy *Flatten* a dále z *Dense* vrstev. Na výstupu jsou 2 neurony ve vrstvě *Softmax*, která zajišťuje klasifikaci do 2 kategorií.[23]

Vrstva *Flatten* je použita jen pro neuronové sítě, které mají na vstupu dvourozměrná data, tedy tréninková data jsou trojrozměrná. To se týká prvního přístupu ke zpracování vstupních dat. U druhého přístupu jsou tréninková data uložena ve dvourozměrném poli, a při učení neuronové sítě jsou jí dvourozměrná data předkládána po řádcích, tedy jako vektory o délce šestnácti hodnot.

Následující ukázka zdrojového kódu ukazuje metodu *create_model* třídy *NNModel*. Počet vrstev typu *Dense* neuronové sítě je určen délkou seznamu *DENSE_NEURONS*, který je proměnnou třídy *Config*. Pomocí tohoto seznamu je možné nastavit počty neuronů v síti, která se má vytvořit. První hodnota seznamu se zadá jako počet neuronů vstupní vrstvy sítě. Další vrstvy jsou do sítě přidávány ve *for* cyklu, který prochází následující počty neuronů v seznamu *DENSE_NEURONS*. Nakonec je přidána výstupní vrstva.

Použití metriky *categorical_accuracy* se v této práci ukázalo jako nevhodné. Při učení neuronové sítě po jednotlivých skupinách souborů[24] se ukázala tato metrika jako nepřesná. U první skupiny souborů dosahovala zobrazovaná přesnost (při procesu učení na tréninkových datech, ne validačních) hodnoty 1.0 (100 %). Při učení na dalších načtených skupinách souborů se zobrazovaná přesnost pohybovala kolem hodnoty 0.46 (46 %). Použití metriky *sparse_categorical_accuracy* tento problém vyřešilo a zobrazovaná přesnost sítě během procesu učení odpovídá skutečné hodnotě.

Jako *optimizer* je použit *Adam*. Vytvořený model je uložen do souboru. Cesta k němu je uložena v class variable *MODEL_PATH* třídy *Config*. Metoda vrací objekt vytvořeného modelu.

[23] Pojmy a základy týkající se tvorby modelů v Keras jsou vysvětleny v kapitole 3.2.
[24] Problematika učení sítě po skupinách souborů je vysvětlena v kapitole 6.4.

```
@staticmethod
def create_model():
    model = Sequential()
    model.add(Dense(input_dim=
                    (Config.TIMESTEPS*Config.NUMBER_OF_SAMPLE_COLUMNS),
                    units=Config.DENSE_NEURONS[0], activation="relu"))
    for number_of_neurons in Config.DENSE_NEURONS[1:]:
        model.add(Dense(units=number_of_neurons, activation="relu"))

    model.add(Dense(units=2, activation="softmax"))
    opt = Adam(learning_rate=0.001)
    model.compile(optimizer=opt,
                  loss="sparse_categorical_crossentropy",
                  metrics=["sparse_categorical_accuracy"])
    print("Model created.")
    model.save(Config.KERAS_MODEL_PATH)
    print("Model saved.")
    return model
```

Každý model neuronové sítě vytvořený pomocí Keras se ukládá do složky, ke které vede z adresáře projektu, kde je uložen soubor main.py, cesta: „./src/models/nazev_modelu/“. V této složce je uložen soubor s příponou .h5, obsahující Keras model a CSV soubory, které se používají při učení a testování neuronové sítě. Dále se do složky, po natrénování modelu na všech souborech, uloží CSV soubor obsahující všechny class variables třídy Config, aby bylo možné se u každého modelu podívat, s jakým nastavením se vytvořil a trénoval. Do této složky se také ukládají soubory modelu TensorFlow Lite[25] a ONNX.

Model je možné exportovat do formátu ONNX metodou *keras_model_to_onnx* třídy *ModelCnversion*.

Pro vytvoření dalšího modelu stačí vytvořit novou složku a cestu k ní uložit do class variable *KERAS_MODEL_PATH* třídy *Config*. Třídy pracují se složkou nastavenou v této proměnné, a je tedy možné volit model, který se má používat, její změnou.

6.4 Učení sítě na vygenerovaných souborech

Soubory, které byly vygenerovány z naměřených dat, zabírají na disku přibližně 108 GB. I přesto, že pro učení neuronové sítě nebyla použita všechna data, která soubory obsahují, bylo potřeba síti předkládat tréninkovou množinu po částech.

Třída *NNModel* obsahuje pouze metody pro vytvoření a načtení neuronové sítě a pro klasifikaci dat. Metody pro učení a vyhodnocení přesnosti sítě obsahuje třída *NNHandler*, která dědí ze třídy *NNModel*. Té se v argumentu konstruktoru předává instance třídy *TrainingSet*. Metody třídy *NNHandler* potom využívají právě tréninková a validační data, které obsahuje instance třídy *TrainingSet*.

Učení po částech je implementováno ve třídě *Trainer*. Nejprve jsou načteny cesty ke všem souborům jako seznam řetězců. Jejich pořadí v seznamu je promícháno, aby každá

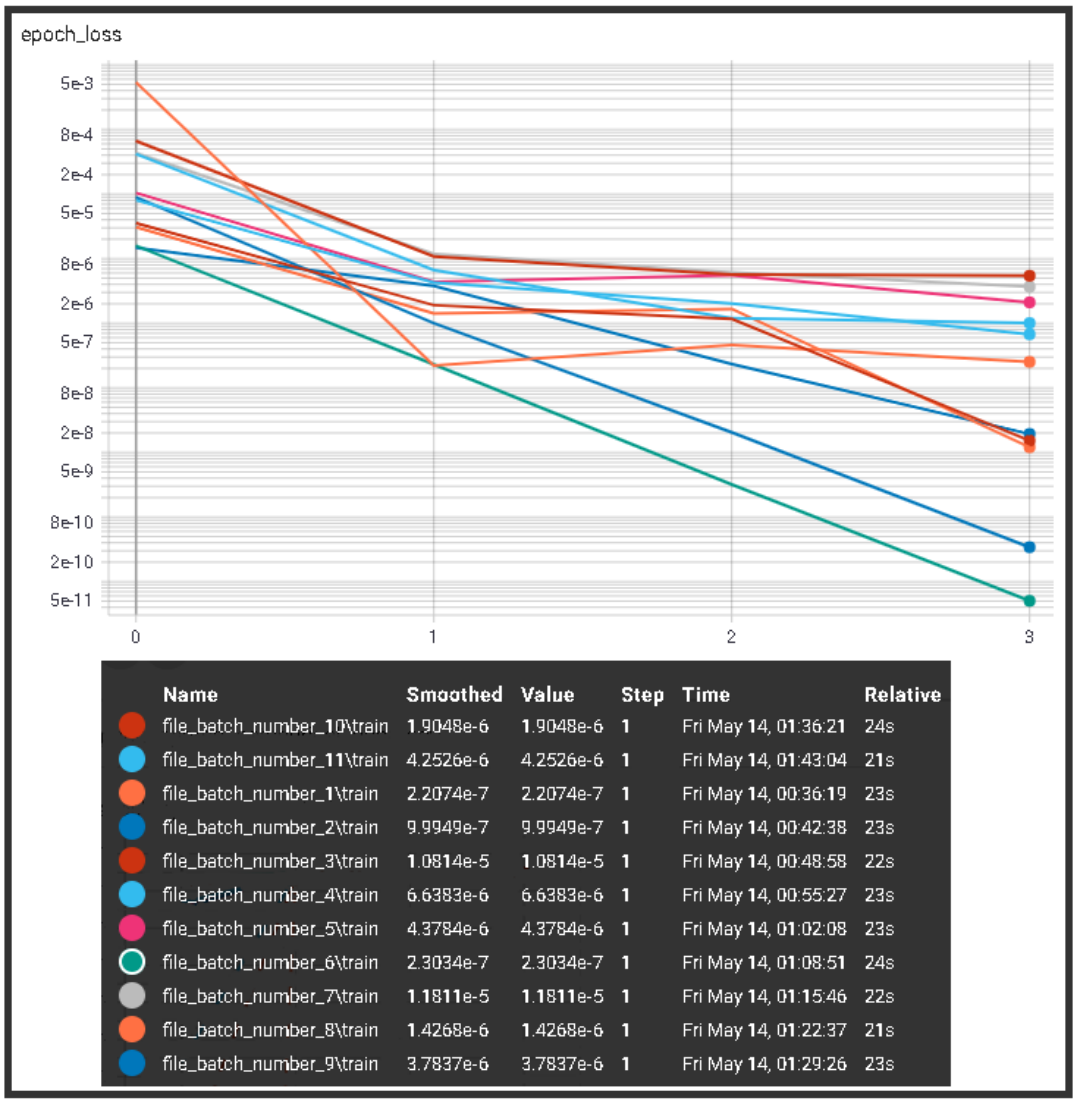
[25] Dále popsáno v kapitole 7.1

skupina souborů, která bude síti předložena k učení, obsahovala data s poruchou i bez poruchy. Metoda *batch_train* prochází seznam cest k souborům, a kontroluje u každé cesty k souboru, jestli už byl daný soubor použit k učení neuronové sítě. Ze seznamu jsou vybírány pouze soubory, které ještě nebyly pro učení použity. Tato kontrola je možná pomocí logovacího CSV souboru s názvem *files_train_log.csv*, který obsahuje cesty k souborům, které už byly pro učení použity. Pro zápis a čtení dat z tohoto souboru slouží třída *TrainLogger*. Cesty k souborům, které ještě nebyly použity při procesu učení, se v seznamu předávají konstruktoru třídy *TrainingSet*, která zajišťuje vytvoření tréninkové a validační množiny. Počet cest k souborům v seznamu, tedy délka tohoto seznamu se nastavuje pomocí class variable *FILE_BATCH* statické třídy *Config*. Tento parametr byl během všech procesů učení v této práci nastaven na hodnotu 35.

Tento algoritmus umožňuje proces učení na všech souborech kdykoli v případě potřeby vypnout a po opětovném spuštění programu pokračovat v učení na souborech, které ještě nebyly použity.

Učení sítě na datech z každé skupiny načtených souborů zajišťuje metoda *train_model* třídy *NNHandler*. Tato data jsou neuronové síti předkládána k učení v takzvaných epochách. Každá epocha znamená, že se data předloží síti k procesu učení jednou. Pro každou načtenou skupinu souborů je síť učena v počtu čtyř epoch (tří epoch pro modely s trojrozměrnými vstupními daty). Počet epoch je nastavitelný pomocí class variable *EPOCHS* třídy *Config*. Učení neuronové sítě s méně než čtyřmi epochami mělo u modelů s dvourozměrnými vstupními daty za následek značně horší průměrnou přesnost. Podobné zhoršení přesnosti se ukázalo při učení sítě při počtu pěti epoch.

Během učení neuronových sítí byl v této práci používán nástroj TensorBoard, který je součástí sady nástrojů TensorFlow a slouží mimo jiné pro vizualizaci veličin, které jsou měřitelné při procesu učení, jako jsou odchylka výstupu sítě a přesnost. Na obrázku 6.3 je zobrazen vývoj odchylky výstupu neuronové sítě při procesu učení. Vývoj odchylky je zobrazen pro všech 11 skupin souborů, do kterých bylo rozděleno všech 384 souborů. Ty jsou na obrázku označeny jako file_batch_number_číslo. Vývoj odchylky je zobrazen zaznamenán v průběhu 4 epoch, během kterých byla neuronová síť učena na každou skupinu souborů.



Obrázek 6.3 Obrázek vývoje odchylky výstupu neuronové sítě v průběhu učení na všech souborech rozdělených do 11 skupin při počtu 4 epoch

6.5 Natrénované modely neuronových sítí

V této práci bylo vytvořeno několik modelů neuronových sítí. Snahou bylo tyto modely otestovat na validačních datech a porovnat jejich přesnost a dobu potřebnou pro klasifikaci vstupních dat. Modely se navzájem liší počtem neuronů v jednotlivých vrstvách a také počtem vrstev. Dalším rozdílem mezi modely je použití paměti v datech, tedy počet *TIMESTEPS* se kterými pracují. Při práci byly používány hodnoty *TIMESTEPS* jedna (tedy data bez přidané paměti) a pět.

6.5.1 Volba způsobu vytváření vstupních dat

Použití dvourozměrných dat jako vstup neuronové sítě se ukázalo z pohledu kompatibility modelu a jednoduchosti vytvoření vstupních dat jako lepší řešení než použití trojrozměrných. Možnost použití trojrozměrných dat tedy ve zdrojových kódech není. Přesnost klasifikace modelů trénovaných na trojrozměrných datech byla o něco lepší než u modelů trénovaných na dvourozměrných. Neuronové sítě se vstupní vrstvou *Flatten* se však ukázaly jako více problematické při snaze je použít v prostředí MATLAB, zvlášť pokud v něm měli být načteny po převodu do formátu ONNX. Výsledky naměřených přesností modelů natrénovaných na trojrozměrných datech jsou v tabulce 6.5.

6.5.2 Validace natrénovaných modelů

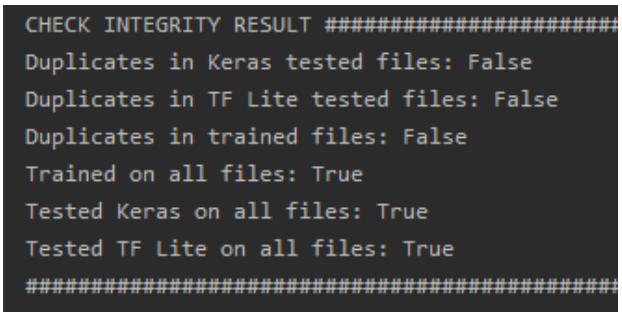
Při procesu validace se testují natrénované modely Keras, a také modely TensorFlow Lite, které jsou vytvořeny z již natrénovaných Keras modelů. Použití TensorFlow Lite je popsáno v kapitole 7.1. Přesnost modelů TensorFlow Lite je však uvedena v této kapitole, protože test přesnosti modelů neuronových sítí se provádí pro Keras i TensorFlow Lite modely současně, aby se nemusely načítat všechny soubory z disku více než jednou.

Měření přesnosti modelů bylo prováděno na všech souborech. Výsledky tohoto měření pro každý natrénovaný model jsou v tabulce 6.1. Pro každý ze 384 souborů byla měřena přesnost klasifikace na validačních datech. Testování modelů zajišťuje metoda *evaluate_per_file* třídy *Tester*. Ta využívá třídy *TestLogger* a *TestLoggerTFLite* pro čtení a zápis do CSV souborů *files_test_log.csv* a *files_test_log_tflite.csv*. Ty slouží pro ukládání výsledků testů do CSV souborů zvlášť pro Keras model a zvlášť pro TensorFlow Lite model. Oba tyto soubory jsou strukturou zapisovaných dat prakticky stejné. Každý z nich obsahuje změřenou přesnost modelu a název souboru, na kterém byla tato přesnost změřena. Spolu s přesností a názvem souboru je v CSV souboru uložena informace v podobě textu „True“ nebo „False“ o tom, zda byl daný soubor použit při procesu učení sítě.

Metoda *evaluate_per_file* provádí testování modelů Keras a TensorFlow Lite na jednotlivých souborech. V této metodě je využívána metoda *get_all_filenames* tříd *TestLogger* a *TestLoggerTFLite* která vrací seznam názvů souborů, které jsou už v CSV souborech uloženy jako otestované. Metoda *evaluate_per_file* pracuje se dvěma seznamy názvů souborů *all_files* a *already_tested_files*. Metoda prochází názvy souborů v seznamu *all_files*, a pokud není název zároveň v seznamu *already_tested_files*, tak je soubor s tímto názvem otestován a následně zapsán jeho název a výsledek testu do CSV souboru. Pro každý načtený soubor je vytvořena instance tříd *NNHandler* a *TFLiteModel*, kterým je předána instance třídy *TrainingSet*, která obsahuje validační data ze souboru.

Pro kontrolu správnosti dat v obou výše zmíněných CSV souborů, a také souboru *files_train_log.csv*, byla naprogramována třída *TestIntegrity*. Její metoda *test_results* po zavolání zkontroluje, zda bylo učení a validace neuronových sítí provedena na všech

dostupných souborech, a jestli CSV soubory neobsahují více záznamů se stejným názvem souboru. Algoritmy učení a validace na souborech jsou navrženy tak, že se jeden soubor nevyskytuje vícekrát v záznamech v CSV souborech. Test na vícenásobný výskyt názvu souboru slouží pro ověření správnosti algoritmů při jejich vývoji. Výstup této metody je text vypsaný do konzole. Obrázek 6.4 zobrazuje vypsaný text v situaci, kdy už proběhl trénink i validace na všech dostupných souborech.



Obrázek 6.4 Obrázek textu vypsaného metodou test_results třídy TestIntegrity do konzole

Pro získání dat uvedených v tabulkách 6.1, 6.5 a 6.4 byla použita metoda *get_nb_of_files_by_acc* tříd *TestAnalysis* a *TestAnalysisTFLite*. Tato metoda pracuje s daty, která jsou uložena v souborech *files_test_log.csv* a *files_test_log_tflite.csv*. Metdoa *get_nb_of_files_by_acc* zjišťuje, u kolika souborů byla dosažena přesnost při validaci v daných intervalech přesnosti. Počet souborů se změřenou přesností validace se započítá do daného intervalu přesnosti, pokud je jejich změřená přesnost menší nebo rovna horní hranici přesnosti v intervalu a zároveň pokud je větší než spodní hranice intervalu. Property *overal_test_accuracy* vrací průměrnou přesnost modelu na validačních datech vypočítanou průměrem z přesností zaznamenaných ve zmíněných CSV souborech.

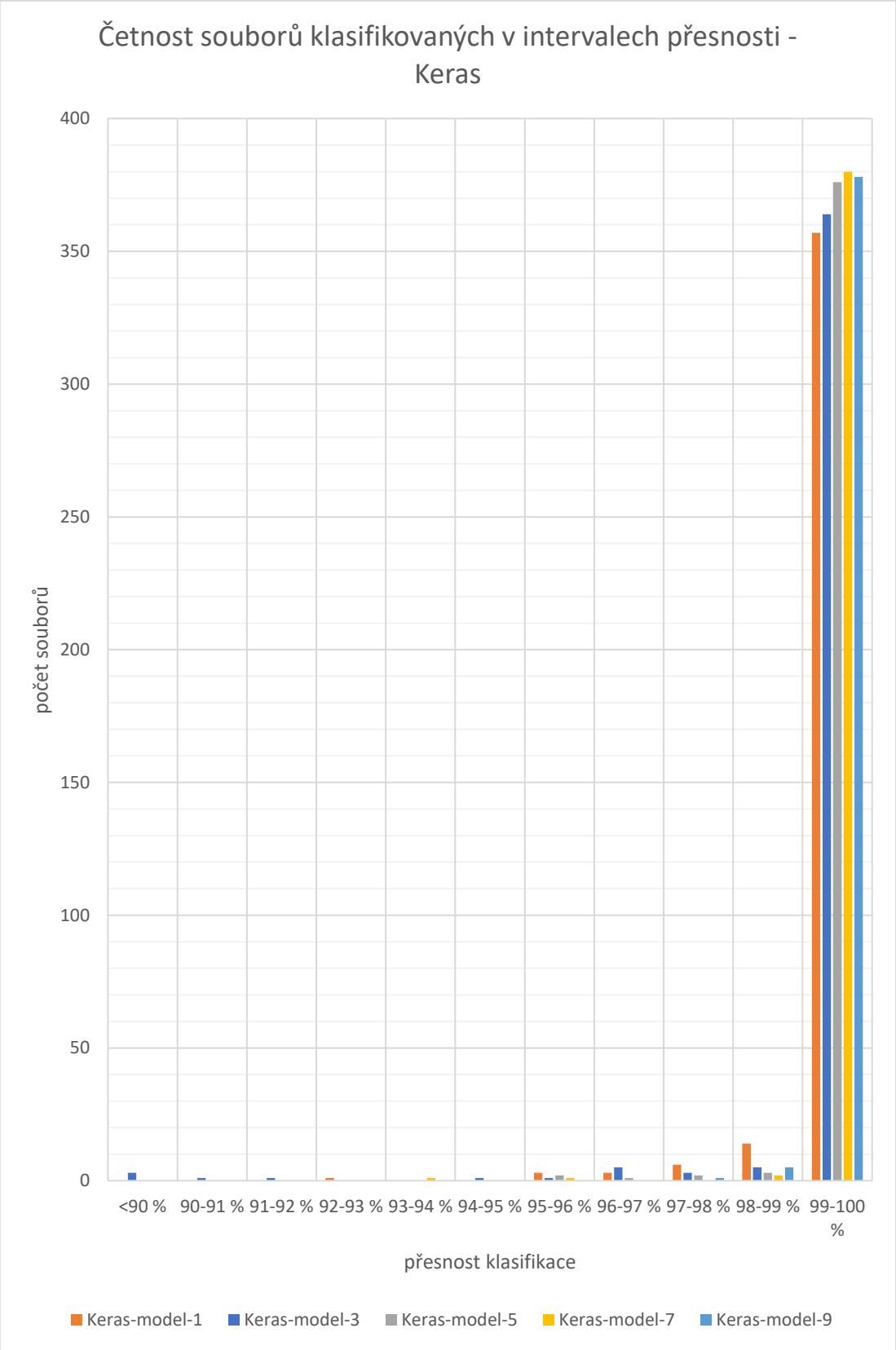
6.5.3 Měření doby potřebné pro klasifikaci vstupních dat neuronové sítě

Pro měření doby, kterou neuronové síti trvá vyhodnotit vstupní data je používána metoda *speed_test*, která je implementována ve třídách *NNHanlder* a *TFLiteModel*. Metoda prochází ve for cyklu validační data v počtu iterací stanovených v class variable *SPEED_TEST_ITERATIONS* třídy *Config*. Měření času je prováděno pomocí rozdílu aktuálních časů zjištěných před a po provedením kódu ve for cyklu. Tyto časy jsou zjištěny pomocí metody *datetime.datetime.now()* pro jejíž použití je potřeba importovat knihovnu *datetime*. Změřený rozdíl se vydělí počtem klasifikovaných vzorků, tedy proměnnou *SPEED_TEST_ITERATIONS* a výsledkem je průměrná doba potřebná pro klasifikaci vstupních dat. Změřené časy natrénovaných modelů implementovaných na zařízení Jetson Nano jsou v tabulce 6.2. Změřené časy těch stejných modelů implementovaných na PC jsou v tabulce 6.3.

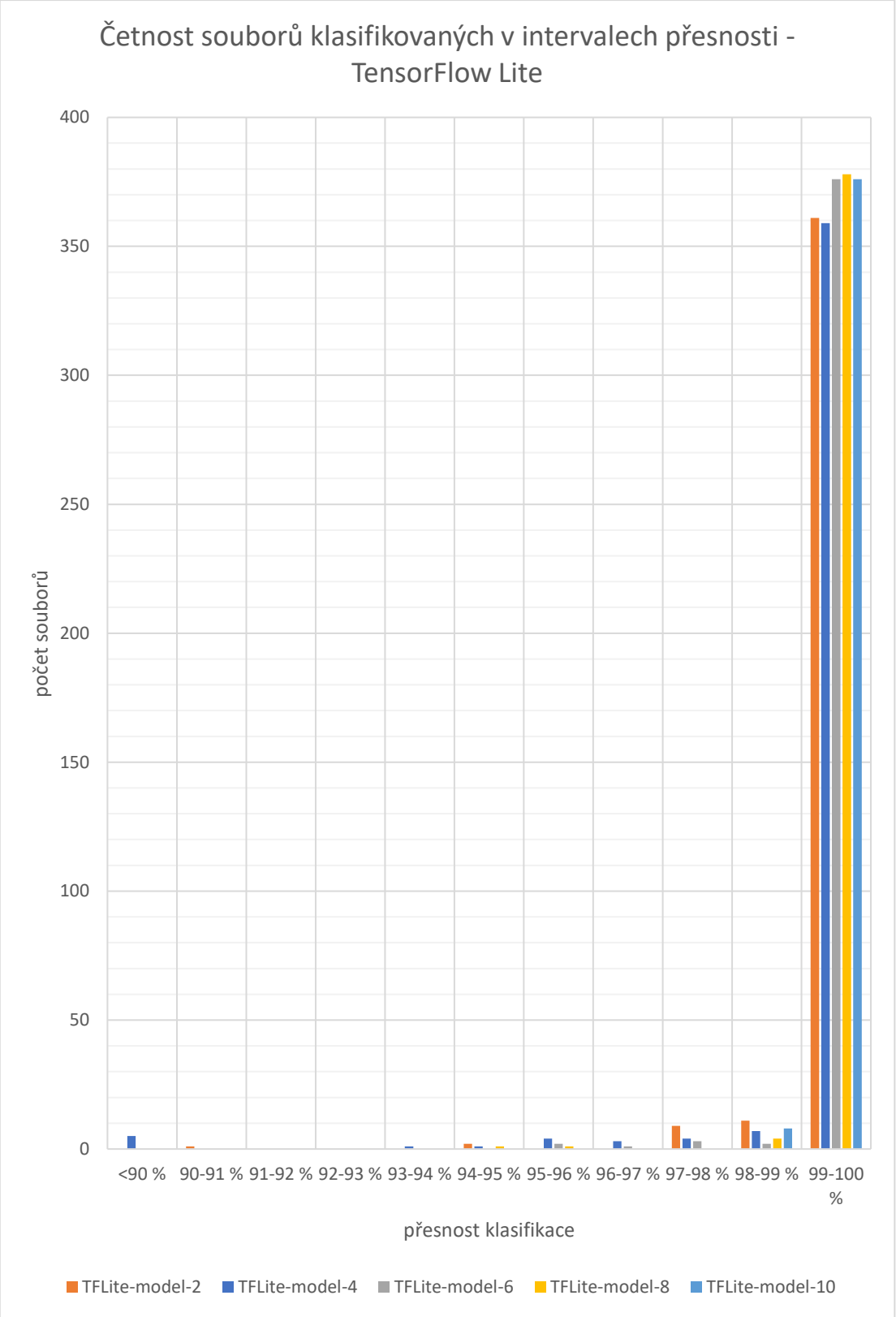
Tabulka 6.1 Tabulka naměřených přesností natrénovaných modelů s dvourozměrnými vstupními daty

				počet souborů klasifikovaných v intervalu přesnosti [-]			
číslo modelu	Použitý nástroj	TIMESTEPS	počty neuronů ve vrstvách	<90 %	90-91 %	91-92 %	92-93 %
1	Keras	1	16-8-2	0	0	0	1
2	TensorFlow Lite	1	16-8-2	0	1	0	0
3	Keras	1	32-16-2	3	1	1	0
4	TensorFlow Lite	1	32-16-2	5	0	0	0
5	Keras	1	64-32-16-2	0	0	0	0
6	TensorFlow Lite	1	64-32-16-2	0	0	0	0
7	Keras	5	80-40-2	0	0	0	0
8	TensorFlow Lite	5	80-40-2	0	0	0	0
9	Keras	5	160-80-2	0	0	0	0
10	TensorFlow Lite	5	160-80-2	0	0	0	0

počet souborů klasifikovaných v intervalu přesnosti [-]							
93-94 %	94-95 %	95-96 %	96-97 %	97-98 %	98-99 %	99-100 %	průměrná přesnost na všech souborech [%]
0	0	3	3	6	14	357	99,7681
0	2	0	0	9	11	361	99,7841
0	1	1	5	3	5	364	99,6501
1	1	4	3	4	7	359	99,5962
0	0	2	1	2	3	376	99,9136
0	0	2	1	3	2	376	99,9043
1	0	1	0	0	2	380	99,9295
0	1	1	0	0	4	378	99,9184
0	0	0	0	1	5	378	99,9280
0	0	0	0	0	8	376	99,9196



Obrázek 6.5 Graf četnosti soborů klasifikovaných v intervalech přesnosti modely Keras. Data jsou z tabulky 6.1



Obrázek 6.6 Graf četnosti soborů klasifikovaných v intervalech přesnosti modely TensorFlow Lite. Data jsou z tabulky 6.1

Tabulka 6.2 Tabulka změřených časů klasifikace modelů implementovaných na zařízení NVIDIA Jetson Nano

			doba klasifikace [µs]	
čísla modelů	TIMESTEPS	počty neuronů ve vrstvách	Keras	TensorFlow Lite
1, 2	1	16-8-2	222235	47
3, 4	1	32-16-2	222491	50
5, 6	1	64-32-16-2	216939	51
7, 8	5	80-40-2	214460	56
8, 9	5	160-80-2	214397	65

Tabulka 6.3 Tabulka změřených časů klasifikace modelů implementovaných na PC

			doba klasifikace [µs]	
čísla modelů	TIMESTEPS	počty neuronů ve vrstvách	Keras	TensorFlow Lite
1, 2	1	16-8-2	33301	18
3, 4	1	32-16-2	32146	29
5, 6	1	64-32-16-2	32561	67
7, 8	5	80-40-2	32152	151
8, 9	5	160-80-2	32137	368

Tabulka 6.4 Tabulka změřené přesnosti klasifikace na vybraných souborech na zařízení NVIDIA Jetson Nano

				počet souborů klasifikovaných v intervalu přesnosti [-]	
číslo modelu	Použitý nástroj	TIMESTEPS	počty neuronů ve vrstvách	<95 %	95-96 %
1	Keras	1	16-8-2	0	1
2	TensorFlow Lite	1	16-8-2	0	1

počet souborů klasifikovaných v intervalu přesnosti [-]				
96-97 %	97-98 %	98-99 %	99-100 %	průměrná přesnost [%]
0	1	1	50	99,8042
0	1	1	50	99,7739

Tabulka 6.5 Tabulka naměřených přesností natrénovaných modelů s trojrozměrnými vstupními daty

				počet souborů klasifikovaných v intervalu přesnosti [-]	
Číslo modelu	Použitý nástroj	TIMESTEPS	počty neuronů ve vrstvách	<93 %	93-94 %
11	Keras	1	16-8-2	0	0
12	TensorFlow Lite	1	16-8-2	0	0
13	Keras	1	32-16-2	0	0
14	TensorFlow Lite	1	32-16-2	0	0
15	Keras	1	64-32-16-2	0	0
16	TensorFlow Lite	1	64-32-16-2	0	0
17	Keras	5	80-40-2	0	1
18	TensorFlow Lite	5	80-40-2	0	1
19	Keras	5	160-80-2	0	0
20	TensorFlow Lite	5	160-80-2	0	0

počet souborů klasifikovaných v intervalu přesnosti [-]						
94-95 %	95-96 %	96-97 %	97-98 %	98-99 %	99-100 %	průměrná přesnost [%]
0	0	0	1	5	378	99,9525
0	0	0	1	7	376	99,9321
0	1	1	1	6	375	99,9301
1	0	0	2	6	375	99,9191
0	0	2	0	0	382	99,9481
0	0	1	1	1	381	99,9418
0	0	0	1	2	380	99,9382
0	0	1	0	3	379	99,9289
1	1	0	0	8	374	99,9271
1	1	0	0	8	374	99,9205

7. OPTIMALIZACE NEURONOVÉ SÍTĚ

Před použitím neuronových sítí ve vestavných zařízeních je vhodné tyto sítě optimalizovat. Pro optimalizaci sítě za účelem dosažení co nejkratší doby potřebné pro klasifikaci předložených dat spolu se zmenšením velikosti sítě v paměti byl použit nástroj TensorFlow Lite.

První krok optimalizace modelu neuronové sítě byl proveden už v rámci kapitoly 6, kde bylo natrénováno více modelů neuronových sítí, za účelem porovnání jejich přesností a rychlostí klasifikace v závislosti na velikosti neuronové sítě. Druhým krokem optimalizace modelů v této práci je jejich kvantizace.

7.1 TensorFlow Lite

TensorFlow Lite je součástí sady nástrojů TensorFlow. Umožňuje převod natrénovaných modelů neuronových sítí z Keras a TensorFlow do úspornějšího formátu, který je vhodný pro implementaci na mobilních a na vestavných zařízeních. V této práci je TensorFlow Lite použit pro kvantizaci natrénovaných Keras modelů.

Pro převod modelů do formátu TensorFlow Lite a práci s nimi byla v kódu vytvořena třída *TFLiteModel*. Jejímu konstruktoru je potřeba při vytváření instance předat instanci třídy TrainingSet, stejně jako u metody *NNHandler*. V konstruktoru této třídy je volána metoda *set_up_interpreter*, která zajišťuje načtení TensorFlow Lite modelu z disku. Pokud není ve složce modelu soubor nalezen, metoda *to_tf_lite* vytvoří nový a následně ho uloží. Pro vytvoření modelu je ale potřeba, aby složka obsahovala již vytvořený Keras model.

7.1.1 Kvantizace modelu

Metoda *to_tf_lite* zajišťuje kvantizaci modelu při jeho vytváření. Pro možnost náhledu na datové typy parametrů uvnitř neuronové sítě po kvantizaci byla vytvořena metoda *export_layers_to_excel*, která parametry jednotlivých vrstev zapíše ve složce modelu do souboru *models_layers.xlsx*.

Snahou bylo kvantizovat neuronové sítě tak, aby jejich parametry byly datového typu int8 a vstupy a výstupy sítě zůstaly v datovém typu float32.

Tabulka 7.1 je zjednodušením tabulky, která se zapisuje do souboru *models_layers.xlsx*. Obsahuje pouze názvy jednotlivých vrstev v modelu, jejich id, rozměr a datový typ parametrů ve vrstvě. V tabulce je vidět, že se vrstvy, kromě těch s *id* 1, 2, 3, 11 a 12 kvantizovaly do datového typu int8. Vrstvy s *id* 1, 2, 3, které obsahují prahy vrstev neuronové sítě jsou kvantizovány do datového typu int32. Pouze vstupní a výstupní vrstva zůstala v datovém typu float32.

Při kvantizaci došlo k významnému zmenšení souborů, ve kterých jsou modely uloženy. Srovnání velikostí souborů původních modelů Keras a konvertovaných modelů

do TensorFlow Lite je v tabulce 7.2. Z dat lze dojít k závěru, že model je po kvantizaci přibližně 8,5 až 11krát menší, než původní Keras model. A to při horší průměrné přesnosti klasifikace v průměru o 0,01334 % oproti Keras modelu[26].

Tabulka 7.1 Tabulka kvantizovaných vrstev TensorFlow Lite modelu č. 2 z tabulky 6.1

id	name	shape	dtype
0	dense_input_int8	[1 16]	<class 'numpy.int8'>
1	sequential/dense/BiasAdd/ReadVariableOp/resource	[16]	<class 'numpy.int32'>
2	sequential/dense_1/BiasAdd/ReadVariableOp/resource	[8]	<class 'numpy.int32'>
3	sequential/dense_2/BiasAdd/ReadVariableOp/resource	[2]	<class 'numpy.int32'>
4	sequential/dense/MatMul	[16 16]	<class 'numpy.int8'>
5	sequential/dense_1/MatMul	[8 16]	<class 'numpy.int8'>
6	sequential/dense_2/MatMul	[2 8]	<class 'numpy.int8'>
7	sequential/dense/Relu;sequential/dense/BiasAdd	[1 16]	<class 'numpy.int8'>
8	sequential/dense_1/Relu;sequential/dense_1/BiasAdd	[1 8]	<class 'numpy.int8'>
9	sequential/dense_2/BiasAdd	[1 2]	<class 'numpy.int8'>
10	Identity_int8	[1 2]	<class 'numpy.int8'>
11	dense_input	[1 16]	<class 'numpy.float32'>
12	Identity	[1 2]	<class 'numpy.float32'>

[26] Vypočteno z průměrné přesnosti v tabulce 6.1.

Tabulka 7.2 Tabulka porovnání velikostí modelů Keras a TensorFlow Lite na disku (modely z tabulky 6.1)

			velikost modelu na disku [kB]	
Čísla modelů	TIMESTEPS	počty neuronů ve vrstvách	Keras	TensorF low Lite
1, 2	1	16-8-2	35	4
3, 4	1	32-16-2	44	4
5, 6	1	64-32-16-2	80	8
7, 8	5	80-40-2	145	13
8, 9	5	160-80-2	335	29

ZÁVĚR

V této práci jsou popsány v současnosti používané způsoby využití umělé inteligence ve vestavných systémech. Je zde popsán jednodeskový počítač NVIDIA Jetson Nano.

V kapitolách 3.2 a 3.3.2 jsou na obrázcích 3.2 a 3.5 uvedeny ukázky kódů, kterými je možné v Keras a Deep Learning Toolbox-u vytvořit jednoduchou neuronovou síť. Tyto obrázky mají za cíl ukázat základní princip použití těchto nástrojů. Je k nim uveden popis základních prvků, které se v daných nástrojích používají pro vytvoření umělých neuronových sítí.

V kapitole 5 je popsána struktura dodaných souborů. Z těch byla vybrána data z matice *DQ_oscilacie_Filtrovanie16,* obsahující 16 hodnot v každém kroku - vzorku. Tato data byla použita pro veškerou práci s neuronovými sítěmi. Zvolená matice dat se ukázala jako vhodná pro naučení neuronové sítě, což je vidět na prudkém poklesu odchylky výstupu neuronové sítě při procesu učení. (obrázek 6.3)

Kapitola 6.1 je věnována vytvoření vstupních dat pro neuronové sítě. Byly vyzkoušeny oba přístupy popsané v této kapitole. Přesnější výsledky klasifikace byly změřeny u modelů využívající trojrozměrná vstupní data. Tyto výsledky jsou zaznamenány v tabulce 6.5. Jako lepší z pohledu implementace modelu mimo napsaný kód, se ukázal přístup využívající dvourozměrná data. Například v prostředí MATLAB je potom jednodušší připravit vstupní data, než při použití trojrozměrných vstupních dat. Při exportu sítě do formátu ONNX nebyl problém ani s modelem obsahujícím vstupní vrstvu typu *Flatten,* ani s modelem využívajícím dvourozměrná data. Při snaze importovat v prostředí MATLAB tyto ONNX modely (pomocí funkce *importONNXNetwor*k), nastala vždy u modelu s *Flatten* vrstvou chyba a nebylo ho možné načíst.

Keras model se neukázal jako dostatečně rychlý na to, aby byl použit v praktické aplikaci pro vyhodnocování poruchy elektromotoru. TensorFlow Lite model sice poskytuje o přibližně 0,01334 % horší průměrnou přesnost oproti Keras modelu, avšak zrychlení predikce po kvantizaci ho dělá pro takovou aplikaci použitelnějším, alespoň co se rychlosti klasifikace vstupních dat týče. Nejjednodušší TensorFlow Lite model, označený v tabulce 6.1 jako model číslo 2, zvládne klasifikovat podle tabulky 6.2 jeden vzorek vstupních dat za 47 µs. To znamená, že je schopný vyhodnotit přibližně 21276 vstupních vzorků za sekundu. V reálném použití takového modelu pro detekci poruchy v elektromotoru by takového počtu klasifikací za sekundu nebylo dosaženo, protože rychlost klasifikace byla změřená při použití už připravených vzorků, které byly uložené v paměti. V reálném čase by se naměřený čas prodloužil o komunikaci a manipulaci s daty, což by nemuselo být významné. Klasifikaci na motoru s periodou PWM 100 µs by pravděpodobně mohlo být možné spouštět v reálném čase.

Model 8 (TensorFlow Lite 80-40-2), využívající přidanou paměť posledních pěti vzorků (včetně aktuálního) má změřenou přesnost klasifikace přibližně o 0,16 % lepší za cenu pomalejší klasifikace o 9 µs oproti nejjednoduššímu modelu.

Zdrojové kódy v jazyce Python musely být přizpůsobeny pro spuštění na zařízení Jetson Nano. Bylo potřeba upravit formát importů naprogramovaných tříd z ostatních souborů a také cesty k používaným adresářům a souborům. Tyto upravené zdrojové kódy jsou v příloze.

Pro srovnání výsledků klasifikace na zařízení NVIDIA Jetson Nano a klasifikace na PC bylo z důvodu omezené kapacity úložiště na Jetson Nano vybráno pouze 53 souborů, na kterých byla provedena validace modelu číslo 1 a 2 – tedy nejjednodušší model Keras a jeho kvantizovaná verze. Výsledek tohoto měření je uveden v tabulce 6.4. Změřená přesnost modelu odpovídá přibližně výsledkům změřeným na PC (tabulka 6.1). Pro měření přesnosti na zařízení Jetson Nano bylo potřeba upravit zdrojový kód v souboru *train_test.py*, protože v kódu přizpůsobeném operačnímu systému Ubuntu jsou použity jiné cesty k souborům, než na PC. Program by tedy vyhodnotil, že na dostupných *MATLAB Data* souborech nebyla neuronová síť učena a byly by použity všechny data ze souborů. Validace by nebyla provedena jen na nezávislých validačních datech, a mohla by být nepřesná. Úprava kódu spočívá v automatickém označení všech dostupných souborů jako těch, na kterých byla síť už naučena. Potom jsou pro validaci použity pouze validační data stejně jako při validaci na PC.

Počítač použitý pro tuto práci byl notebook Dell Inspiron 15 7577 s procesorem Intel Core i5-7300HQ, 16 GB RAM a grafickou kartou NVIDIA GeForce GTX 1060 Max-Q. V příloze je v souboru *„seznam_pouzitych_knihoven_python.txt“* uveden seznam použitých knihoven jazyka Python, které byly při práci používány.

LITERATURA

[1] SAWHNEY, Mohanbir. Why Apple And Microsoft Are Moving AI To The Edge. *Forbes* [online]. 2020, 27.1.2020 [cit. 2021-01-03]. Dostupné z: https://www.forbes.com/sites/mohanbirsawhney/2020/01/27/why-apple-and-microsoft-are-moving-ai-to-the-edge/?sh=1e326b112570

[2] WU, Jun. Edge AI Is The Next Wave of AI. *Towards Data Science* [online]. 2020, 19.4.2020 [cit. 2021-01-03]. Dostupné z: https://towardsdatascience.com/edge-ai-is-the-next-wave-of-ai-a3e98b77c2d7

[3] ROJEK, Marcin. Artificial Intelligence: on-device or in the cloud or data center? *Becoming Human: Artificial Intelligence Magazine* [online]. 2018, 28.6.2018 [cit. 2021-01-03]. Dostupné z: https://becominghuman.ai/artificial-intelligence-on-device-or-in-the-cloud-or-data-center-701d03527abb

[4] *Post-training quantization* [online]. 2020 [cit. 2021-01-03]. Dostupné z: https://www.tensorflow.org/lite/performance/post_training_quantization

[5] NVIDIA CORPORATION. *Buy the Latest Jetson Products* [online]. 2021 [cit. 2021-01-03]. Dostupné z: https://developer.nvidia.com/buy-jetson

[6] NVIDIA CORPORATION. *JETSON NANO* [online]. 2020 [cit. 2021-01-03]. Dostupné z: https://www.nvidia.com/cs-cz/autonomous-machines/embedded-systems/jetson-nano/

[7] NVIDIA CORPORATION. *JetPack SDK* [online]. [cit. 2021-01-03]. Dostupné z: https://developer.nvidia.com/EMBEDDED/Jetpack

[8] GIGA-BYTE TECHNOLOGY CO., LTD. *CUDA* [online]. [cit. 2021-01-03]. Dostupné z: https://www.gigabyte.com/Glossary/cuda

[9] MULESOFT LLC, A SALESFORCE COMPANY. *What is an API? (Application Programming Interface)* [online]. [cit. 2021-01-03]. Dostupné z: https://www.mulesoft.com/resources/api/what-is-an-api

[10] *API Documentation* [online]. 2020 [cit. 2021-01-03]. Dostupné z: https://www.tensorflow.org/api_docs

[11] KURKURE, Uday. Machine Learning on VMware vSphere 6 with NVIDIA GPUs. *VMware* [online]. 2016, 6.10.2016 [cit. 2021-01-03]. Dostupné z: https://blogs.vmware.com/performance/2016/10/machine-learning-vsphere-nvidia-gpus.html

[12] CHOLLET, François. *The Sequential model* [online]. 2020 [cit. 2021-01-03]. Dostupné z: https://keras.io/guides/sequential_model/

[13] *Flatten layer* [online]. [cit. 2021-01-03]. Dostupné z: https://keras.io/api/layers/reshaping_layers/flatten/

[14] *Layer activation functions* [online]. [cit. 2021-01-03]. Dostupné z: https://keras.io/api/layers/activations/

[15] SARKAR, Kanchan. ReLU : Not a Differentiable Function: Why used in Gradient Based Optimization? and Other Generalizations of ReLU. *Medium* [online]. 2018, 31.5.2018 [cit. 2021-01-03]. Dostupné z: https://medium.com/@kanchansarkar/relu-not-a-differentiable-function-why-used-in-gradient-based-optimization-7fef3a4cecec

[16] Optimizers [online]. [cit. 2021-01-03]. Dostupné z: https://keras.io/api/optimizers/

[17] BROWNLEE, Jason. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. *Machine Learning Mastery* [online]. 2017, 3.7.2017 [cit. 2021-01-03]. Dostupné z: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[18] *Probabilistic losses* [online]. [cit. 2021-01-03]. Dostupné z: https://keras.io/api/losses/probabilistic_losses/

[19] *Accuracy metrics* [online]. [cit. 2021-01-03]. Dostupné z: https://keras.io/api/metrics/accuracy_metrics/

[20] *Deep Learning Toolbox™* [online]. [cit. 2021-01-03]. Dostupné z: https://www.humusoft.cz/matlab/deep-learning/

[21] *Deep Learning Toolbox* [online]. [cit. 2021-01-03]. Dostupné z: https://www.mathworks.com/help/deeplearning/

[22] THE MATHWORKS, INC. *AnalyzeNetwork* [online]. [cit. 2021-01-03]. Dostupné z: https://www.mathworks.com/help/deeplearning/ref/analyzenetwork.html

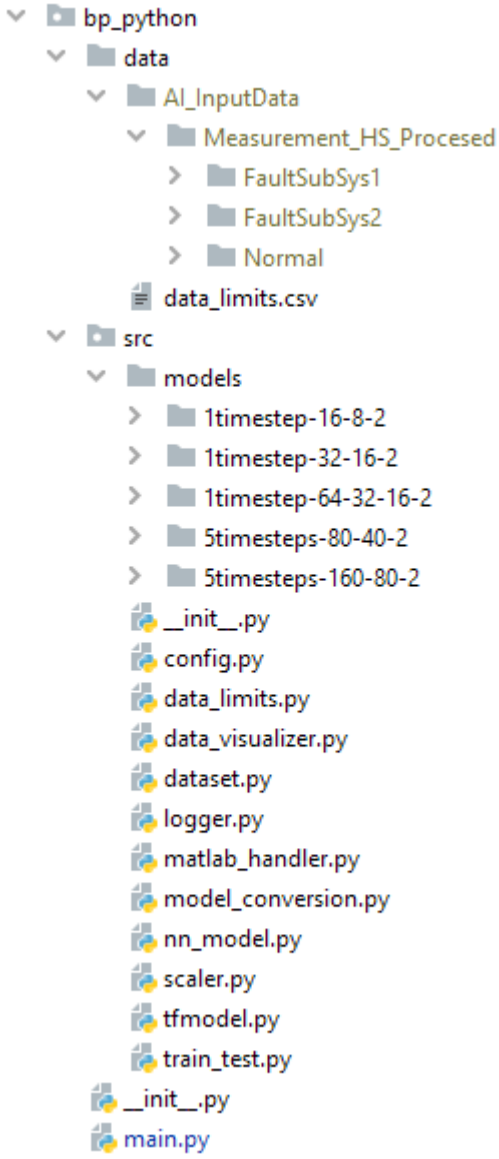
[23] NVIDIA CORPORATION. *Getting Started with Jetson Nano Developer Kit* [online]. [cit. 2021-01-03]. Dostupné z: https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

[24] NVIDIA CORPORATION. *DEEP LEARNING FRAMEWORKS DOCUMENTATION* [online]. 2020 [cit. 2021-01-03]. Dostupné z: https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html

[25] GPU Coder*: Generate CUDA code for NVIDIA GPUs* [online]. [cit. 2021-5-23]. Dostupné z: https://www.mathworks.com/products/gpu-coder.html

SEZNAM PŘÍLOH

Příloha A - Obrázek adresářové struktury



Obrázek 0.1 Obrázek adresářové struktury vytvořeného Python projektu včetně složky *AI_InputData* s dodanými daty

Příloha B - Zdrojové kódy na přiloženém DVD

Zdrojové kódy na přiloženém DVD obsahují i několik dodaných souborů. Verze příloh odevzdaná do informačního systému tyto soubory z důvodu jejich velikosti neobsahuje.

Adresářová struktura zdrojových kódů v jazyce Python je zobrazena na obrázku 0.1 v příloze A.

/…….Kořenový adresář přiloženého DVD

zdrojove_kody.zip……Komprimovaný soubor se zdrojovými kódy

bp_python_jetson……..…Zdrojové kódy pro Jetson Nano (stejná struktura jako bp_python)

bp_python……...…….Složka se zdrojovými kódy v jazyce Python

main.py……..……..……..……spouštěcí Python soubor

data……..……..…….....Složka s daty

AI_InputData…….....……..…Složka se složkami s dodanými soubory

data_limits.csv……..…....…..CSV soubor s min a max hodnotami dat

src……..……..……..*Python package* se zdrojovými kódy

models……...…Složka obsahující složky modelů Keras a TensroFlow Lite

1timestep-16-8-2……..…….Složka modelu č. 1 a 2

1timestep-32-16-2……...…….……..…..……Složka modelu č. 3 a 4

1timestep-64-32-16-2……..…..……..……..…Složka modelu č. 5 a 6

5timestep-80-40-2….……..……..…..……..…Složka modelu č. 7 a 8

5timestep-160-80-2……..……..…Složka modelu č. 9 a 10

__init__.py.....Python __init__.py soubor pro Package

config.py.....Python soubor se třídou *Config*

data_limits.py.....Python soubor se třídou *DataLimits*

data_visualizer.py…..... Python soubor se třídou *DataVisualizer*

dataset.py.....Python soubor se třídami *Dataset* a *TrainingSet*

logger.py.....Python soubor se třídami pro práci s CSV soubory

matlab_handlert.py.....Python soubor se třídou *MatlabData*

model_conversion.py.....Python soubor se třídou *ModelConversion*

nn_model.py.....Python soubor se třídami *NNModel* a *NNHandler*

scaler.py……..…..……...Python soubor se třídou *Scaler*

tfmodel.py……..…...Python soubor se třídou *TFLiteModel*

train_test.py…..…Python soubor se třídami *Trainer*, *Tester* a *TestIntegrity*

bp_matlab…….……Složka se zdrojovými soubory pro MATLAB

data_to_csv.m……..MATLAB skript pro převod dat do CSV

gpu_setup_check.m…..…MATLAB skript pro kontrolu instalovaných nástrojů

jetsonScipt.m……..……..MATLAB skript pro vygenerování GPU Coder-em
gpu_setup_check.m…..…MATLAB skript pro kontrolu instalovaných nástrojů
prevod_keras_do_mat.m……MATLAB skript pro export Keras modelu
main.cu…….CUDA main soubor pro vygenerování spustitelného souboru ELF
main.h…..…CUDA main soubor pro vygenerování spustitelného souboru ELF